Windows 2000 프로그람작성법

교육성 프로그람교육쎈터

차 례

제 4 장. 차림표의 기초

제 5 장. 대화칸

제 6 장. 여러가지 조종체

제 7 장. 비트매프의 사용법과 다시그리기문제의 해결방법

제 8 장. 본문의 조종

제 9 장. 도형의 사용법

제 10 장. 공통조종체와 공통대화칸

제 11 장. 오르내리기조종체, 추적띠 및 진행띠

제 12 장. 상태창문, 표쪽조종체 및 나무구조보기조종체

제 13 장. 특성표와 조수

제 14 장. 머리부조종체와 월사업표조종체

제 15 장. 스레드에 기초한 다중과제처리

제 16 장. 두 종류의 도움말체계

제 17 장. 인쇄기의 사용법

제 18 장. 체계자료기지의 리용법과 화면보호기의 작성

제 19 장. 차림표의 확장

제 20 장. 동적련결서고와 보안

머 리 말

Windows 2000 은 콤퓨터를 최적조건에서 동작시키기 위해 Microsoft 가 21 세기를 지향하여 개발한 조작체계이다. Windows 2000 은 최신의 32bit 콤퓨터기술의 표준으로서 자기의 위력을 충분히 발휘하고 있다. Windows 2000 은 프로그람전문가들이 기대하고 바라오던 대단히 도전적인 프로그람개발환경이다. 물론 이 도전을 받아 들이면 커다란 리익이 있다.

Windows 2000 은 충분히 조정되고 엄격하게 설계된 쏘프트웨어이다. 범용성이 있으며 강력하고 사용하기 쉽다는 우점이 내부의 복잡성을 은페하고 있다. 체계의 안정성과 미리 갖추어 진 보안기능은 이 조작체계가 기업관련의 사용자들도 만족시키게 한다. 물론 많은 쏘프트웨어개발자들도 같은 리유로 하여 Windows 2000 을 리용하고 있다.

이 책은 독자들에게 Windows 2000 프로그람을 작성하는 방법을 가르치기 위한 도서로서 기초적인것으로부터 시작하여 중요한 문제들에 이르기까지 모두 포괄하고 있다. 또한 고급한 프로그람작성기술들도 많이 소개한다.

만일 독자들이 Windows 프로그람을 작성해 본 경험이 없다면 이 책을 제 1 장부터 차례로 읽어 나가야 한다. 다음 장의 내용은 앞장에서 설명한 내용을 기초로 하여 전개되여 있기때문에 부분적으로 뛰여 넘으며 읽어서는 안된다.

Windows 95 등의 다른 판본의 Windows 프로그람을 작성해 본 경험이 있다면 이 책을 보다 빨리 읽어 나갈수 있을것이다. 그러나 기초지식을 설명하는 장들에는 주의를 돌려야 한다. 그것은 Windows 2000 과 다른 판본의 Windows 사이에 중요한 차이점이 일부 존재하기때문이다.

이 책을 리용하는데 필요한 프로그람작성의 기초지식

이 책을 효과적으로 리용하려면 C 혹은 C++에 의한 프로그람작성경험이 필요하다. 그것은 C 혹은 C++가 Windows 2000 의 프로그람개발언어이기때문이다. 만일 C 나 C++에 대한 지식의 부족을 느낀다면 기초지식의 기반을 공고히 하는데 시간을 투하해야 한다.

Windows 2000 환경에서 프로그람을 개발하는것은 구조체와 공용체, 지시자 등을 활용하는것으로 된다. 이러한것들에 습관되여 있지 않다면 Windows 2000 프로그람을 작성하는것이 곤난하다.

이 책을 리용하는데 필요한 쏘프트웨어

이 책에서 제시하는 원천코드들을 번역하려면 Windows 2000 프로그람을 작성하는 기능을 가진 C++번역프로그람이 있어야 한다. 이 책에서는 Microsoft Visual C++6.0 에서 원천코드의 번역결과를 확인하였다.(일부 프로그람을 제외하고는 Borland C++ Compiler 5.5 로도 번역할수 있다)

이 책의 실례들은 표준적인 C++의 문법규칙에 맞추어 작성한 프로그람들이므로 파일이름의 확장자로 .cpp 를 사용하여야 한다. 그러나 거의 모든 부분에서 C++의 C 언어기능만이 사용되고 있으므로 응용프로그람을 C 나 C++의 임의의 언어로 작성한다고 해도 이 책의 원천코드들을 그대로 사용할수 있다.

이 책의 프로그람을 번역할 때에는 Windows 프로그람을 작성하기 위한 체계설정을 리용하여야 한다. 그러나 MFC 와 같은 클라스서고의 지원은 선택하지 말아야 한다. 즉 Microsoft Visual C++를 사용한다면 새로운 프로젝트에서 Win32 Application 을 선택해야 한다.

제 1 장

Windows 2000 의 기초

Windows 2000 은 Microsoft 의 최신판 32bit 조작체계이다. 이 책에서는 Windows 2000 프로그람을 작성하는 방법을 서술한다. 프로그람작성의 기초도서인 이 책은 Windows 2000 의 기능들중에서 프로그람작성과 직접 관련이 없는 부분에는 많은 페지를 할당하지 않는다. 그대신 될수록 빨리 Windows 2000 프로그람을 작성할수 있도록 실리적인 방법을 주고 있다. 우선 Windows 2000 프로그람을 파악하기 위한 첫 단계로서 Windows 2000 의 주요기능들에 대하여 설명한다.

Windows 2000 은 매우 거대하고 복잡한 조작체계이므로 한권의 책에서 그의 모든 기능을 설명할수는 없다. 그러므로 광범히 리용되는 기능들과 중요한 새 기능들중에서 모든 Windows 2000 프로그람에 공통적인 항목들만을 취급하며 모든 Windows 2000 프로그람작성자들이 반드시 알아야 할 내용들을 포괄한다. 이 책은 중요한 조작체계인 Windows 2000 을 보다 깊이 학습하기 위한 기초지식을 주도록 서술되여 있다.

Windows 2000 이란 무엇인가?

Windows 2000 은 Windows NT 에 기초하고 있는 Microsoft 의 본격적인 32bit Windows 가동환경(Platform)이지만 그보다 한 단계 더 발전한 조작체계이다. NT 기술에 기초하고 있지만 Windows 2000 은 새롭게 고쳐 설계하고 코드도 고쳐 써 NT 에 비하여 보다 강력하면서도 유연성을 가지게 되였다. 례하면 내장된 보안기능과 확장된 망기능 및 최신하드웨어에 대한 대응 등 새로운 기능들을 가지고 있다. Windows 2000 은 Windows 를 21 세기에로 인도하는 조작체계이다.

Windows 2000 에는 다음과 같은 네가지 환경이 있다.

- Windows 2000 Professional
- Windows 2000 Server
- Windows 2000 Advanced Server
- Windows 2000 Datacenter Server

이 책의 내용은 우에서 서술한 모든 환경에 대응하고 있다. 기본적인 프로그람작성수법은 모든 환경에서 다 같다. 많은 리용자들은 프로그람을 작성하거나 검사하는데 Windows 2000 Professional 을 사용하고 있다. *Windows 2000 Professional* 은 Windows NT 4.0 Workstation 의 갱신판이며 결과적으로는 말단사용자를 위한 Windows 2000 이다.

Windows 2000 은 완전한 32bit 조작체계이다. 이것은 Windows 95 에서와 같이 32bit 와 16bit 가 뒤섞인 코드를 사용할수 없다는것을 의미한다. 순수한 32bit 체계를 제공하는것으로 하여 Windows 2000 에는 낡은 조작체계를 괴롭혀 오던 결함이나 문제점들이 배제되였다.

또한 Windows 2000 전용으로 서술된 프로그람만이 아니라 Windows 95/98/NT 용 프로그람의 대부분을 실행할수 있다. 그밖에도 Windows 2000 은 DOS 프로그람을 비롯한 종래의 16bit 프로그람도 실행할수 있다. Windows 2000 은 실행하는 프로그람에 알맞는 적절한 환경을 자동적으로 제공해 준다. 례를 들어 DOS 프로그람을 실행하려는 경우에는 Windows 2000 이 16bit DOS 의 가상기계와 프로그람을 실행하기 위한 지령대기상태창문을 자동적으로 작성해 준다.

Windows 2000 은 C2 보안준위에 해당한 보안기능을 실현하였다. 이 보안준위는 암호에 의한 등록가입(Log on), 소유권에 의한 자원호출제한과 동작내용을 기록하는 기능(Log)을 제공한다. 기억기는 사용되기전에 초기화되며 프로그람에 의하여 사용되는 기억기는 보호되여 다른 프로그람의 영향을 받지 않는다. 따라서 한 프로그람이 다른 프로그람을 파괴하거나 그의 내용을 참조할수 없도록 되여 있다.

응용프로그람대면부

프로그람작성자의 관점으로는 Windows 2000 이 프로그람사이의 대면부(Interface)를 정의하고 있는것처럼 보인다. 모든 응용프로그람은 이 대면호출을 통하여 Windows 2000 과 련계를 취한다. Windows 2000 대면호출은 조작체계의 기능을 호출하도록 정의되여 있는 일련의 함수모임으로 되여 있다. 이 함수들을 통털어 *응용프로그람대면부(API : Application Programming Interface)*라고 한다.

API 는 수백개의 함수들을 포괄하고 있으며 응용프로그람은 기억기의 할당과 화면에 대한 출력 및 창문의 작성과 같은 조작체계가 가지고 있는 기능이 필요될 때 API 를 호출하여 리용한다. API 가운데는 도형장치대면부(*GDI : Graphics Device Interface*)라는것이 있다. 이것은 장치에 의존하지 않는 도형을 제공하는 Windows 기능의 일부이다.

자주 사용되는 API 에는 기본적으로 *Win16* 과 *Win32* 의 두가지 부류가 있다. Win16 은 낡은 16bit 판 API 이다. Win32 는 현재 주류로 되고 있는 32bit 판 API 이다. Win16 은 Windows 3.1 에서 사용되였다. Windows 2000 프로그람은 Win32 를 사용한다. 일반적으로 Win32 는 Win16 의 확장묶음이라고 한다. 사실 많은 함수들은 꼭 같으며 사용법도 같다.

그러나 기능이나 사용법이 류사하다고 하여도 Win16 과 Win32 에는 기본적으로 두가지 큰 차이가 있다. 첫번째 차이는 Win32 가 32bit 의 련속적인 주소공간을 지원하고 있는 반면에 Win16 은 16bit 토막을 사용하는 기억기모형을 지원한다는것이다. 이 차이는 Win16 에서 16bit 파라메터를 사용하고 있는 부분에 Win32 에서는 32bit 파라메터를 사용한다는것을 의미한다.

두번째 차이는 Win32 에는 스레드(Thread)에 의한 다중과제, 보안 및 기타 Win16 에는 없는 확장기능을 지원하는 API 가 있다는것이다. 만일 Windows 프로그람을 작성해 본 경험이 없다면 이 차이를 인식할 필요는 없을것이다. 그러나 이전에 작성했던 16bit 의 코드를 Windows 2000 에 이식하려고 한다면 매 API 함수에 주는 파라메터들을 충분히 확인해 보아야 한다.

참고 *: Win16 API 는 DOS 준위에서 동작하는 Windows 3.1 응용프로그람과의 호환성때문에 아직까지 사용되고 있다. 그러나 Windows 2000 프로그람에서는 반드시 Win32 API 를 사용해야 한다.*

동적련결서고

Win32 의 API 함수는 *동적련결서고*(*DLL : Dynamic Link Library*)라고 부르는 파일에 들어 있다. DLL 의 기능은 매개 프로그람이 실행될 때 호출된다. API 함수는 DLL 에 재배치가능한 형식으로 넣어 져 있다. API 함수를 호출하는 프로그람을 번역하는 시점에서 련결프로그람은 프로그람의 실행(EXE)파일에 함수코드를 써넣지 않는다. 그대신에 DLL 의 이름이나 함수이름 등 함수의 적재정보만을 써넣는다.

프로그람을 실행하면 필요한 API 코드가 Windows 2000 에 의하여 적재된다. 이런 방식에 의하여 매 프로그람에 API 코드를 포함시키거나 그것을 디스크에 보관할 필요가 없어지게 된다. API 코드는 실행시에 프로그람이 기억기에 적재될 때 추가된다.

DLL 을 리용하는데는 매우 중요한 여러가지 우점이 있다.

첫째로, 사실상 모든 프로그람이 API 함수를 사용하는것으로 되므로 DLL 이 차지하는 디스크공간을 절약할수 있다. 만일 디스크에 넣어 진 매 프로그람의 EXE 파일에 API 함수가 직접 추가된다면 대량의 중복된 코드로 하여 디스크공간이 소비되게 된다.

둘째로, Windows 2000 자체의 갱신이나 확장을 DLL 파일을 변경하는것만으로 간단히 실현할수 있으며 이미 작성된 프로그람들을 다시 번역할 필요는 없다.

셋째로, DLL 을 사용함으로서 다른 종류의 조작체계의 모의를 쉽게 실현할수 있다.

다중스레드에 의한 다중과제의 지원

Windows 2000 은 *다중과제*조작체계이다. 이것은 두개이상의 프로그람을 동시에 실행할수 있다는것을 의미한다. 물론 CPU 가 한개인 체계에서는 개개의 프로그람이 CPU 를 공유하므로 그것들을 완전히 동시에 실행할수는 없다. Windows 2000 은 두개이상의 CPU 를 장비한 콤퓨터에서도 동작한다. 이 경우에는 프로그람을 완전히 동시에 실행할수 있다. 그러나 작성한 프로그람이 실행되는 콤퓨터의 종류를 언제나 확인할수 있는것은 아니므로 동시실행은 항상 고려해야 할 과제정도에서 생각하는것이 좋다.

Windows 2000 은 프로세스준위 및 스레드준위 두 종류의 다중과제처리를 지원한다. *프로세스*란 실행중의 프로그람을 말한다. Windows 2000 은 프로세스준위의 다중과제를 지원하고 있으므로 동시에 한개이상의 프로그람을 실행할수 있다. Windows 2000 이 지원하는 프로세스준위의 다중과제는 고전적인 수법이다.

Windows 2000 이 지원하고 있는 두번째 다중과제수법은 스레드준위의 다중과제처리이다. *스레드*란 관리와 실행이 가능한 코드단위이다. 이 이름은 [실행줄(Thread of execution)]이라는 고찰법으로부터 붙여 진것이다. 모든 프로세스는 적어도 하나의 스레드를 가진다. 그러나 Windows 2000 프로세스는 여러개의 스레드를 가지는것이 보편적

이다.

Windows 2000 이 스레드를 다중과제로서 실행할수 있다는것과 개개의 프로세스가 한개이상의 스레드를 가지고 있다는데로부터 한개의 프로세스는 동시에 실행되는 두개이상의 스레드를 가질수 있게 된다. 그러므로 Windows 2000 을 사용하면 프로그람을 다중과제로 실행할수 있을뿐아니라 한 프로그람에서 매 부분을 다중과제로 실행할수도 있다. 그러나 이러한 우월성을 살리자면 상당히 수준 있는 프로그람을 작성하여야 한다.

여러가지 파일체계

Windows 2000 에 대하여 또 한가지 알아 두어야 할것은 아래에 표시한 여러가지 *파일체계*를 지원한다는것이다.

- FAT(File Allocation Table)
- FAT32(FAT 를 32bit 로 확장한것)
- NTFS(NT File System)

FAT 파일체계는 DOS 용으로 설계된 낡은 파일체계이다. 이 파일체계는 장기간에 걸쳐 사용되고 널리 보급되였다. 그러나 FAT 파일체계는 DOS 와의 호환성을 유지한다는 리유로 하여 용량에 제한이 있다. FAT 의 제한성은 *FAT32* 에 의하여 개선되였으며 *FAT32* 는 대용량의 디스크를 지원한다. *NTFS* 는 Windows NT 전용으로 설계되였으며 Windows 2000 에서 개량이 가해 진 수법이다. *NTFS*에는 보안기능 등 수많은 우점이 있다. 어느 파일체계를 사용하여도 문제가 없다.

제 2 장

Windows 2000 프로그람 작성의 기초

이 장에서는 Windows 2000 프로그람작성의 기초를 학습한다. 이 장의 첫째 목적은 프로그람이 Windows 2000 과 련계를 취하는 방법과 모든 Windows 2000 응용프로그람에 공통되는 규칙을 학습하는데 있다. 둘째 목적은 다른 장들에서 작성하는 Windows 2000 프로그람의 시초로 되는 *골격코드*를 작성하는것이다. 모든 Windows 프로그람에는 공통된 부분이 있으며 이것은 이 장에서 작성하게 되는 응용프로그람의 골격코드에 들어 있다.

먼저 Windows 2000 프로그람을 작성하기 위한 두가지 방법을 설명해 보자.

Windows 2000 프로그람을 작성하기 위한 두가지 방법

Windows 2000 프로그람을 작성하는데는 두가지 방법이 있다. 첫번째 방법은 Win32가 제공하는 *API 함수*를 리용하는것이다. 이 방법에서는 프로그람에서 API 를 직접 리용하며 따라서 Windows 2000 프로그람의 모든 요소를 상세히 서술하게 된다.

두번째 방법은 API 함수들을 은페시킨 C++ *클라스서고*를 사용하는것이다. 지금까지 널리 보급되여 온 Windows 프로그람을 위한 클라스서고는 *MFC(Microsoft Foundation Classes)*이다. MFC 나 Windows 용의 다른 클라스서고들은 여러가지 측면에서 많은 도움을 주는 강력한 개발도구이다. 그러나 이것들은 API 를 사용한 Windows 프로그람작성방법의 기초를 명백히 알지 못하면 원만하게 사용할수 없다. 그 리유는 다음과 같다.

첫째로, 모든 Windows 프로그람에 공통되는 기본적인 구성방식이 있다는것이다. MFC 는 이 구성방식의 대부분을 은페하고 있다. Windows 구성방식에 대한 지식은 장기적인 프로그람작성을 계획하고 있다면 중요한 지식으로 된다. 례하면 이 지식이 Windows 프로그람에 대한 오유수정을 쉽게 해준다.

둘째로, API 를 사용함으로써 프로그람의 동작을 세밀하면서도 완전하게 조종할수 있기때문이다.

셋째로, 모든 Windows 2000 프로그람개발환경들이 API 에 의한 개발을 지원하고 있기때문이다. API 의 지식은 여러가지 측면에서 리용할수 있다.

넷째로, API 를 리용하여 Windows 2000 프로그람을 작성할수 있으면 MFC 나 다른 클라스서고도 쉽게 정통할수 있기때문이다. 즉 API 를 리용한 프로그람작성방법을 모르는 사람은 우수한 Windows 프로그람작성자라고 말할수 없다.

Windows 2000 프로그람의 외형과 조작성

Windows 2000(Windows 전반)의 목적에는 사전에 어떤 련습이 없이도 누구든지 체계앞에 앉으면 간단히 응용프로그람을 조작할수 있어야 한다는 큰 전제가 있다.

이로부터 Windows 는 통일적인 대면부를 사용자에게 제공한다. 리상적으로는 하나의 Windows 프로그람을 조작해 본 경험이 있으면 다른 모든 Windows 프로그람을 조작할수 있도록 되여야 할것이다.

물론 대다수의 실제 프로그람들은 그것을 효과적으로 리용하려면 약간한 훈련이 필요하다. 그러나 적어도 이 훈련의 목적은 사용자가 프로그람을 어떻게 조작하는가가 아니라 프로그람자체의 약간한 기능을 파악하는데 있다. 사실 Windows 응용프로그람코드의 대부분은 *사용자대면부*를 지원하기 위한것으로 되여 있다.

Windows 2000 에서 동작하는 모든 프로그람은 창문형식의 대면부를 자동적으로 제공하는것이 아니라는것을 알아야 한다. Windows 는 일관성을 주장하기 위한 환경을 제공하지만 그것을 강요하는것은 아니다. 례하면 표준적인 Windows 형식의 대면부를 구성요소로 하지 않는 Windows 프로그람을 작성할수도 있다.

Windows 형식의 프로그람을 작성하려면 이 책에서 설명하는 기술을 리용하여 정확한 수법에 따라야 한다. Windows 를 사용하도록 서술된 프로그람만이 일반적인 Windows 프로그람의 외형과 조작성을 가질수 있다.

기본적인 Windows 설계의 기본원칙을 무시하려는 생각이 있다면 그렇게 해야 할 리유가 필요하다. 왜냐하면 그러한 프로그람은 사용자대면부를 통일한다는 Windows 가 큰 전제로 내세우고 있는 목적을 무시하는것으로 되기때문이다. 일반적으로 Windows 2000 응용프로그람을 작성하는 경우 표준적인 Windows 형식의 규칙과 설계수법에 따라야 한다.

여기서는 Windows 2000 응용프로그람의 조작성을 결정하는 중요한 요소들을 몇가지 보기로 하자.

탁상모형

약간 례외는 있지만 창문형식의 사용자대면부의 목적은 화면에 탁상면과 꼭같은 환경을 제공하는것이다. 탁상에는 여러가지 종류의 서류가 있고 그것들이 겹놓일수 있다. 현재 보이는 제일 꼭대기의 서류밑에는 다른 서류가 있다. Windows 2000 에서 탁상면에 해당한것은 현시장치의 화면이다. 다종다양한 서류에 해당한것은 화면상의 창문이다. 탁상에서는 서류의 위치를 이동시킬수 있을뿐아니라 임의의것을 제일 우에 놓을수 있고 눈으로 볼수 있는 범위를 확대할수도 있다. Windows 2000 은 창문에 대해서 같은 조작을 할수 있도록 되여 있다. 창문을 선택하여 그것을 현재의 창문으로 할수 있다. 이것은 그 창문을 다른 창문우에 놓는다는것을 의미한다. 창문의 크기를 크게 하거나 작게 할수도 있으며 화면상에서의 위치를 이동시킬수도 있다. 즉 Windows 는 탁상우에서의 조작과 같은 방법으로 화면을 조작할수 있다. 일반적인 프로그람에서 이러한 조작들이 가능하다.

마우스

다른 판본의 Windows 에서와 같이 Windows 2000 에서도 대부분의 조작과 선택 및 화면작업 등에 *마우스*를 사용한다. 물론 건반도 사용할수 있지만 Windows 에서는 마우스를 사용하는것이 더 편리하다. 그러므로 앞으로 작성하는 프로그람에서도 입력장치로 될수록 마우스를 지원해야 한다. 차림표선택, 흘림띠의 조작 등을 포함한 일반적인 조작에 마우스가 자동적으로 지원된다.

아이콘, 비트매프 및 도형

Windows 2000 에서는 *아이콘*, *비트매프* 및 기타 종류의 *도형*을 사용할수 있다. *아이콘*은 어떤 조작, 자원 또는 프로그람을 상징적으로 나타내는데 사용되는 작은 그림기호이다. 비트매프는 사용자에게 정보를 간단하면서도 재빠르게 전달하기 위하여 사용되는 4 각형의 도형이다. 비트매프가 차림표의 요소로 사용되는 경우도 있다. Windows 2000 은 직선, 4 각형, 원 등을 그리는 기능을 비롯해서 수많은 도형기능을 지원한다. 이러한 도형기능들을 적절히 사용하는것은 Windows 프로그람을 완성하기 위한 중요한 고리이다.

차림표, 조종체, 대화칸

Windows 는 사용자의 입력을 받아 들이기 위한 여러가지 표준적인 항목들을 제공한다. 그것들은 *차림표*와 다양한 종류의 *조종체* 및 대화칸 등이다. 간단히 말해서 차림표는 사용자가 선택할수 있는 추가선택항목들을 표시한것이다. 차림표는 Windows 프로그람의 표준항목이므로 API 에 차림표선택기능이 들어 있다. 따라서 프로그람에서 차림표에 대한 조작을 상세하게 서술할 필요는 없다.

조종체는 특정한 형식으로 사용자와의 대화를 가능하게 하는 특수한 창문이다. 례를 들어 누름단추, 흘림띠, 편집칸, 검사칸 등이 있다. 차림표의 사용과 마찬가지로 Windows 에 정의되여 있는 조종체의 조작은 거의 자동화되여 있다. 상세한 조작을 프로그람에 서술하지 않고 조종체를 사용할수 있다.

*대화칸*은 차림표보다 복잡한 조작성을 응용프로그람에 제공해 준다. 례하면 파일이름을 입력하는데 대화칸을 사용할수 있다. 대화칸은 주로 조종체를 배치하기 위하여 사용한다. 대체로 차림표형식을 제외한 입력은 대화칸으로 실현한다.

창문의 구성요소

Windows 2000 프로그람구조를 설명하기에 앞서 창문의 구성요소들의 이름을 설명해보자.(그림 2-1)

모든 창문에는 창문의 테두리를 표시하는 경계선이 있으며 그것을 사용하여 창문의 크기를 변경할수 있다. 창문의 웃부분에는 여러개의 요소가 있다. 왼쪽 웃모서리에는 *체계차림표아이콘*이 있다. 이 아이콘을 찰칵하면 체계차림표가 현시된다. 체계차림표아이콘의 오른쪽에는 창문의 제목이 있다. 오른쪽끝에는 최소화단추, 최대화단추 및 닫기단추가 있다. *의뢰자구역*은 프로그람이 실제로 사용하는 부분이다. 대부분의 창문에는 창문에 표시되는 본문 등을 흐르게 하기 위한 수평흘림띠와 수직흘림띠가 있다.

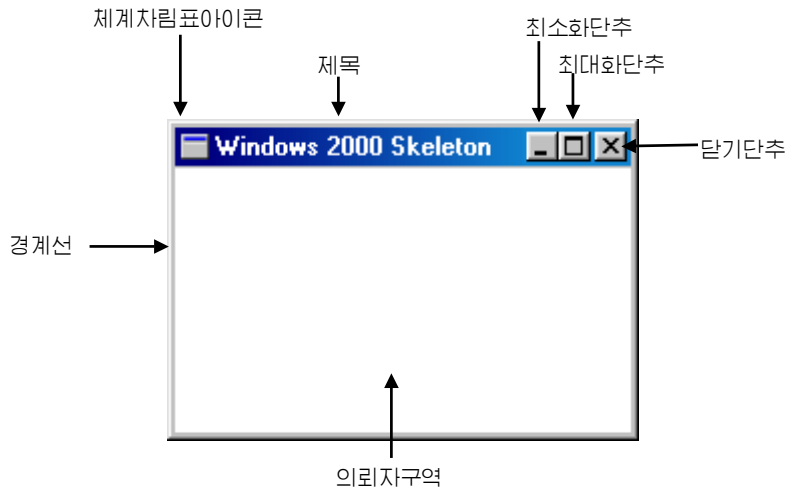



그림 2-1. 표준적인 창문의 구성요소

Windows 와 프로그람이 정보를 주고받는 방법

이전의 조작체계와 프로그람들에서는 조작체계와의 련계를 개시하는것이 프로그람쪽이였다. 례하면 DOS 프로그람에서 입력이나 출력을 요구하는것은 프로그람이다. 결국 고전적방식으로 서술된 프로그람은 조작체계를 호출하며 조작체계가 프로그람을 호출하지는 않는다.

그런데 Windows 에서는 완전히 다른 형식이 리용된다. 프로그람을 호출하는것은 Windows 이다. 이것은《프로그람은 Windows 가 보내오는 통보문을 받는다》는것으로 실현된다. 통보문은 Windows 가 호출하는 특수한 함수에 의하여 프로그람에 전송된다. 통보문이 보내지면 프로그람이 그 내용에 따라서 처리를 진행한다.

통보문에 응답할 때 몇가지 API 함수를 호출할수도 있지만 주도권을 가지고 있는것은 어디까지나 Windows 이다. 이것이 Windows 2000 을 비롯한 모든 Windows 프로그람에서 리용하고 있는 통보문에 의한 통신방식이다.

Windows 2000 이 프로그람에 보내는 통보문에는 여러가지 종류가 있다. 례하면 프로그람창문에서 마우스를 찰칵할 때마다 프로그람에 *마우스찰칵통보문*을 보낸다. 프로그람창문의 크기가 변경되면 또 다른 통보문을 보낸다. 입력초점을 가진 프로그람에서 건반이 눌리울 때마다 또 다른 통보문을 보낸다.

한가지 명백히 알아 두어야 할것은 프로그람과 관련한 통보문들이 임의로 발송된다는것이다. 이로부터 Windows 프로그람은 새치기구동방식의 프로그람과 류사하다고 말할수 있다. 즉 다음에 어떤 통보문을 받게 될지 미리 알수 없다.

Windows 2000 응용프로그람의 기반

WinMain( )

모든 Windows 2000 프로그람의 실행은 *WinMain( )*함수의 호출로부터 시작된다.(Windows 프로그람에는 main( )함수가 없다.) WinMain( )에는 응용프로그람에서 사용되는 다른 함수와는 다른 여러가지 특징이 있다. 우선 WinMain( )함수는 *WINAPI* 호출규약을 지정하여 번역하여야 한다. 프로그람에서 사용되는 함수에는 암시적으로 C의 호출규약이 사용되지만 이것을 다른 호출규약을 사용하도록 변경하여 함수를 번역할 수도 있다.

례하면 자주 사용되는 호출규약으로서 *PASCAL* 호출규약이 있다. 몇가지 리유로 하여 Windows 2000 이 WinMain( )을 호출할 때는 WINAPI 호출규약이 사용된다. 또한 WinMain( )의 돌림값형은 옹근수형(int)형으로 되여야 한다.

이식과 관련한 요점 : 낡은 16bit 판 Windows 프로그람에서는 WinMain( )호출규약으로 PASCAL 이 사용되였다. 이런 응용프로그람들을 Windows 2000 에 이식하는 경우 이것을 WINAPI 호출규약으로 변경하여야 한다.

창문수속(창문함수)

모든 Windows 프로그람은 그 프로그람내부에서가 아니라 Windows 에 의하여 호출되는 특수한 함수를 가질 필요가 있다. 이 함수를 일반적으로 *창문수속* 혹은 *창문함수*라고 부른다. 이 함수를 통하여 Windows 2000 과 프로그람과의 련계가 이루어 진다.

창문함수는 프로그람에 통보문을 보낼 필요가 있을 때 Windows 2000 에 의하여 호출된다. 이때 창문함수의 파라메터에 통보문이 주어 진다. 모든 창문함수는 돌림값형으로서 *LRESULT CALLBACK* 를 지정하여 선언된다. LRESULT 는 32bit 의 부호 없는 옹근수형이다. CALLBACK 는 이 함수가 Windows 로부터 호출된다는것을 나타낸다. Windows 용어에서는 Windows 로부터 호출되는 함수를 *역호출함수*라고 한다.

이식과 관련한 요점 : 낡은 16bit 판 Windows 코드에서는 창문함수를 LONG FAR PASCAL 형으로 정의하였다. Windows 2000 에 이식하는 경우에는 이것을 LRESULT CALLBACK 로 변경하여야 한다.

창문함수가 Windows 2000 이 보내는 통보문을 받으면 그 통보문에 의해 지시되는 처리를 해야 한다. 전형적인 창문함수의 내부는 매개 통보문에 응답하도록 처리를 분기시키는 *switch* 문으로 구성된다.

그러나 프로그람에 보내오는 통보문을 모두 처리할 필요는 없다. 왜냐하면 Windows 2000 에 암시적인 처리를 의뢰할수 있기때문이다. Windows 가 생성하는 통보문의 종류에는 수백가지가 있으므로 대부분의 통보문은 프로그람에서 처리하지 않고 Windows 에 맡기는것이 일반적이다.

모든 통보문은 32bit 의 부호 없는 옹근수값이다. 또한 모든 통보문에는 통보문을 보충하는 몇가지 정보가 부가되여 있다.

창문클라스

Windows 2000 프로그람의 실행을 시작할 때는 처음에 *창문클라스*의 정의와 등록을 진행하여야 한다.(여기서 클라스라는 용어는 C++클라스가 아니라 단순히 양식이나 형식이라는 의미에서 사용된다.) 창문클라스를 등록할 때 Windows 에 창문의 형식과 창문함수를 알려 준다. 그러나 창문클라스를 등록하는것만으로는 창문이 작성되지 않는다. 창문을 작성하려면 앞으로 여러단계가 필요하다.

통보문순환고리

이미 설명한바와 같이 Windows 2000 은 통보문을 보내여 프로그람과 련계를 취한다. 모든 Windows 프로그람은 WinMain( )함수내에 *통보문순환고리*를 포함하고 있다.

이 순환고리에서는 응용프로그람통보문대기렬에 쌓여 있는 통보문을 꺼내여 그것을 Windows 에 되돌려 주면 Windows 가 그것을 파라메터에 넣어 프로그람의 통보문함수를 호출한다. 이와 같은 방식은 통보문을 전달하는 수단으로서 모든 Windows 프로그람이 정확히 동작하게 하는 방식이다.(이 방식을 사용하는 리유의 하나는 일정짜기프로그람(scheduler)가 CPU시간을 할당하는데서 Windows 에 조종을 돌려 주는 편이 응용프로그람의 시간구획이 끝나기를 기다리는것보다 더 합리적이기때문이다.)

Windows 에서 사용하는 자료형

Windows API 함수에서는 int 나 char 와 같은 표준적인 C/C++자료형을 사용하는 경우가 드물다. 그대신에 Windows 에서 사용되는 대부분의 *자료형*은 *WINDOWS.H* 파일 또는 그와 관련한 파일안에 형정의(*typedef*)문으로 정의된 특수한 자료형으로 되여 있다. 이 파일은 Microsoft(및 Windows 용 C++번역프로그람제작자)에 의하여 제공되는 것이며 모든 Windows 프로그람에 포함되여 있어야 한다.

자주 사용되는 자료형으로서 *HANDLE*, *HWND*, *BYTE*, *WORD*, *DWORD*, *UINT*,

LONG, *BOOL*, *LPSTR*, *LPCSTR* 등이 있다. *HANDLE* 은 *손잡이*로서 사용되는 32bit 의 부호 없는 옹근수값이다. 손잡이의 형식에는 여러가지 종류가 있다. 이것들은 모두 HANDLE 과 같은 크기를 가진다. 손잡이란 어떤 자원을 식별하기 위한 단순한 값이다. 례하면 HWND는 창문의 손잡이로서 사용되는 32bit의 부호 없는 옹근수이다. 손잡이를 표시하는 자료형은 모두 H로 시작하는 이름으로 되여 있다.

BYTE 는 8bit 의 부호 없는 옹근수이며 *WORD* 는 16bit 의 부호 없는 옹근수이다. *DWORD*와 UNIT는 32bit 부호 없는 옹근수이고 LONG은 32bit의 부호 있는 옹근수이다.

그리고 *BOOL* 은 옹근수이며 참과 거짓을 나타내는데 사용한다. 그러나 표준적인 C++의 bool과는 달리 BOOL에는 모든 옹근수값을 보관할수 있다. LPSTR는 문자렬에 대한 지시자이며 LPCSTR는 문자렬에 대한 상수(const)지시자이다.

지금까지 설명한 기본적인 자료형외에도 Windows 2000 에 독자적으로 정의되여 있는 여러가지 구조체가 있다. 륜곽코드에서 사용하는 구조체는 MSG 와 *WNDCLASSEX* 이다. MSG 구조체는 Windows 2000 통보문을 보관하기 위한것이고 WNDCLASSEX 구조체는 창문클라스를 정의할 때 사용되는것이다. 이 구조체들에 대해서는 이 장의 뒤부분에서 상세히 설명한다.

이식과 관련한 요점 : UINT는 Windows 2000 프로그람을 번역할 때는 32bit의 부호 없는 옹근수로 되며 16bit 의 Windows 코드를 번역할 때는 16bit 의 부호 없는 옹근수로 된다. 낡은 응용프로그람을 Windows 2000 에 이식하는 경우에는 변경이 필요하다.

Windows 2000 의 골격코드

이제 한 걸음씩 최소한의 Windows 2000 응용프로그람을 작성해 보자. 모든 Windows 2000 프로그람에는 공통적인 부분이 있다. 여기에서는 필수적인 기능과 요소를 가진 최소한의 Windows 2000 *응용프로그람의 골격코드*를 작성하기로 한다.

Windows 프로그람작성분야에서는 응용프로그람의 골격이 재리용된다. 최소한의 프로그람의 크기가 다섯행정도인 DOS 프로그람과는 달리 Windows 2000 에서는 그 크기가 50 행정도로 된다. 이때문에 Windows 프로그람작성에는 응용프로그람의 골격이 재리용된다.

최소한의 Windows 2000 프로그람은 WinMain( )과 창문함수를 가진것으로 된다. WinMain( )함수에서는 다음의 순서로 처리를 진행한다.

- 창문클라스에 기초하여 창문을 작성한다.
- 창문클라스를 정의한다.

- 창문클라스를 Windows 2000 에 등록한다.
- 창문을 표시한다.
- 통보문순환고리의 실행을 개시한다.

창문함수는 프로그람의 동작에 관계되는 모든 통보문에 응답해야 한다. 이 륜곽코드에서는 창문을 표시하는것외에는 아무것도 하지 않으므로 응답해야 할 통보문은 사용자가 프로그람을 완료한다는것을 응용프로그람에 알리는것만으로 된다.

자세한 설명을 하기에 앞서 아래의 프로그람을 실행해 보자. 이것은 최소한의 Windows 2000 프로그람골격이다. 이 골격코드는 제목, 체계차림표단추, 최소화단추, 최대화단추, 닫기단추를 가진 표준창문을 표시한다. 따라서 이 창문은 최소화, 최대화, 이동, 크기변경 및 완료기능을 가지게 된다.

실례 2-1. Skel 프로그람

```
// Windows 2000 의 최소골격코드

#include <windows.h>

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);

char szWinName[] = "MyWin"; // 창문클라스의 이름

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;

  // 창문클라스를 정의한다.
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst;          // 실체의 손잡이
  wcl.lpszClassName = szWinName; // 창문클라스의 이름
  wcl.lpfnWndProc = WindowFunc;  // 창문함수
  wcl.style = 0;                 // 암시적형식
```

```
wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
wcl.hIconSm = NULL; // 큰 아이콘에 대응한 작은 아이콘을 사용한다.
wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

wcl.lpszMenuName = NULL; // 클라스차림표는 없다.
wcl.cbClsExtra = 0;       // 보조기억기령역은 필요 없다.
wcl.cbWndExtra = 0;

// 창문의 배경색을 흰색으로 한다.
wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

// 창문클라스를 등록한다.
if(!RegisterClassEx(&wcl)) return 0;

/* 창문클라스를 등록하였으므로
    창문을 작성할수 있다. */
hwnd = CreateWindow(
  szWinName, // 창문클라스의 이름
  "Windows 2000 Skeleton", // 형식
  WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
  CW_USEDEFAULT, // X 자리표는 Windows 가 결정한다.
  CW_USEDEFAULT, // Y 자리표는 Windows 가 결정한다.
  CW_USEDEFAULT, // 창문의 너비는 Windows 가 결정한다.
  CW_USEDEFAULT, // 창문의 높이는 Windows 가 결정한다.
  NULL,          // 어미창문은 없다.
  NULL,          // 차림표는 없다.
  hThisInst,     // 실체의 손잡이
  NULL           // 추가파라메터는 없다.
);

// 창문을 표시한다.
ShowWindow(hwnd, nWinMode);
UpdateWindow(hwnd);

// 통보문순환고리를 작성한다.
while(GetMessage(&msg, NULL, 0, 0))
{
```

```
        TranslateMessage(&msg); // 건반통보를 변환한다.
        DispatchMessage(&msg);  // Windows 2000 에 조종을 돌려 준다.
    }
    return msg.wParam;
}

/* 이 함수는 Windows 2000 으로부터 호출되며
   통보문대기렬에서 꺼낸 통보문을 받아 들인다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
    switch(message) {
        case WM_DESTROY: // 프로그람을 완료한다.
            PostQuitMessage(0);
            break;
        default:
            /* 이 switch 문에서 지정되지 않은 통보문은
               Windows 2000 에 처리를 맡긴다. */
            return DefWindowProc(hwnd, message, wParam, lParam);
    }
    return 0;
}
```

프로그람의 내용을 차례로 설명해 보자.

우선 모든 Windows 2000 프로그람은 WINDOWS.H 라는 머리부파일을 가져야 한다. 이 파일 및 그와 관련된 파일들에는 API 함수의 선언, 다양한 *자료형*, *마크로* 및 Windows 2000 에서 사용되는 정의가 포함되여 있다. 레하면 HWND 나 WNDCLASSEX 등은 WINDOWS.H 및 그와 관련된 파일들에 정의되여 있다.

이 프로그람에서 사용하는 창문함수의 이름은 WindowFunc( )이다. 이 함수를 사용하여 Windows 2000 이 프로그람과 련계를 취하므로 역호출함수로 된다.

프로그람의 실행은 WinMain( )으로부터 시작된다. WinMain( )에는 4 개의 파라메터가 있다. hThisInst 와 hPrevInst 는 손잡이들이다. *hThisInst* 는 현재 프로그람의 실체를 가리킨다. *hPrevInst* 는 마지막으로 기동된 같은 프로그람의 실체를 나타낸다.

Windows 2000 은 다중과제처리체계이므로 어떤 프로그람의 실체를 동시에 여러개 실행시킬수 있다. 그러나 Windows 2000 에서는 hPrevInst 값이 항상 NULL 로 되여 있다. *lpszArgs* 는 응용프로그람의 기동시에 지정되는 지령행파라메터문자렬에 대한 지시자

이다. Windows 2000 에서는 여기에 프로그람의 이름을 포함한 완전한 지령행문자렬을 보관한다. *nWinMode*에는 프로그람의 기동시에 프로그람을 어떠한 형식으로 표시하는가를 가리키는 값이 보관된다.

WinMain( )안에서는 세개의 변수가 선언된다. hwnd 는 프로그람창문의 손잡이를 보관하기 위한 변수이다. *msg* 는 통보문을 보관하기 위한 구조체형변수이며 *wcl* 는 창문클라스를 정의하기 위한 구조체형변수이다.

이식과 관련한 요점 : 이미 설명한바와 같이 Windows 2000 프로그람에서는 hPrevInst 값이 항상 NULL 로 설정되여 있다. 16bit 응용프로그람에서는 같은 프로그람실체가 실행중이면 hPrevInst 값이 NULL 로 되지 않는다. 이것은 16bit 의 Windows 와 Windows 2000 의 기본적인 차이점의 하나이다. 16bit 환경에서는 같은 프로그람의 여러개의 실체가 창문클라스나 여러가지 자료를 공유한다. 이때문에 응용프로그람은 체계에서 다른 실체가 실행중인가 아닌가를 아는것이 중요하게 된다. 그러나 Windows 2000 에서는 매개 프로세스는 다른 프로세스와 독립이므로 창문클라스를 자동적으로 공유하지 않는다. Windows 2000 에 아직 hPrevInst 파라메터가 남아 있는것은 오직 호환성때문이다.

창문클라스의 정의

*WinMain( )*에서 처음으로 진행하는 두가지 처리는 *창문클라스*를 정의하고 그것을 등록하는것이다. 창문클라스는 WNDCLASSEX 구조체의 매 성원들을 설정하여 정의한다. 매개 성원들은 다음과 같다.

```
UINT cbsize;                    // WNDCLASSEX 구조체크기
UINT style;                     // 창문형태
WNDPROC lpfnWndProc;            // 창문함수주소
int cbClsExtra;                 // 클라스보조기억기령역
int cbWndExtra;                 // 창문보조기억기령역
HINSTANCE hInstance;            // 실체손잡이
HICON hIocn;                    // 큰 아이콘손잡이
HICON hIocnsm;                  // 작은 아이콘손잡이
HCORSOR hCursor;                // 마우스유표손잡이
HBRUSH hbrBackground;           // 배경색
LPCSTR lpszMenuName;            // 기본차림표의 이름
LPCSTR lpszClassName;           // 창문클라스의 이름
```

류곽코드에서 볼수 있는것처럼 cbSize에는 WNDCLASSEX구조체의 크기가 설정된다. hInstance에는 hThisInst가 가리키는 현재 실체의 손잡이가 설정된다. 창문클라스의 이름은 lpszClassName에 설정되며 여기에서는 "MyWin"이라는 문자렬로 되여 있다.

창문함수의 주소는 lpfnWndProc 에 설정된다. 이 프로그람에서는 창문의 형식이 정의되여 있지 않다. 보조기억기령역도 불필요한것으로 되여 있다. 기본차림표도 지정되여 있지 않다. 대부분의 프로그람에는 차림표가 있지만 이 골격코드에서는 사용하지 않는다.

모든 Windows 응용프로그람에서는 마우스유표와 응용프로그람아이콘에 암시적형태를 설정하여야 한다. 이 자원은 응용프로그람자체에서 작성할수도 있고 골격코드에서 하는것처럼 미리 갖추어 진 자원에서 선택할수도 있다. 어느 경우에나 자원의 손잡이를 WNDCLASSEX 구조체의 해당한 성원에 설정하여야 한다. 우선 아이콘의 설정방법을 설명해 보자.

Windows 2000 응용프로그람은 두개의 아이콘을 가지고 있다. 하나는 큰 아이콘이고 다른 하나는 작은 아이콘이다. 작은 아이콘은 응용프로그람이 최소화될 때 사용되며 체계차림표의 위치에도 표시된다. 큰 아이콘은 응용프로그람을 이동 또는 복사할 때 표시된다.

일반적으로 큰 아이콘은 32 ×32의 비트매프이며 작은 아이콘은 16 × 16의 비트매프이다. 큰 아이콘은 *LoadIcon()*라는 API함수로서 적재된다. 아래에 이 함수의 선언을 보여 주었다.

```
HICON LoadIcon(HINSTANCE hInst, LPCSTR lpszName);
```

이 함수는 호출이 성공하면 아이콘의 손잡이를 돌려 주며 실패하면 NULL 을 돌려준다. hInst 에는 아이콘을 포함하는 모듈의 손잡이를 설정하며 아이콘의 이름은 lpszName 에 설정한다. 그러나 Windows 에 미리 갖추어 진 아이콘을 사용하는 경우에는 첫 파라메터에 NULL 을 설정하고 다음 파라메터에 아래에 준 마크로중의 어느 하나를 설정한다.

아이콘 마크로	형 태
IDI_APPLICATION	암시적인 아이콘
IDI_ERROR	오유아이콘
IDI_INFORMATION	정보아이콘
IDI_QUESTION	의문부호아이콘
IDI_WARNING	감탄부호아이콘
IDI_WINLOGO	Windows 의 략호

아이콘을 적재할 때 두가지의 중요한 문제를 고려해야 한다. 첫째로, 응용프로그람에서 작은 아이콘을 지적하지 않으면 큰 아이콘의 자원파일내용이 검사된다. 만일 자원파일안에 작은 아이콘도 들어 있다면 그 아이콘이 사용된다. 그렇지 않은 경우에는 작은 아이콘이 요구될 때 큰 아이콘의 축소판이 사용된다. 작은 아이콘을 지정하지 않는 경우에는 골격코드에서 진행하는것처럼 hIconSm 값에 NULL 을 설정하여야 한다.

둘째로, 큰 아이콘을 적재하는데는 LoadIcon( )을 사용하는것이 일반적이다. *LoadImage( )*를 사용하면 다른 크기의 아이콘을 넣을수 있지만 이 책에서는 대체로 Windows에 미리 갖추어 진 아이콘만을 사용하고 있으므로 Windows 2000이 큰 아이콘과 작은 아이콘을 다 제공해 준다.

마우스유표를 적재하기 위해서 *LoadCursor( )*라는 API 함수를 사용한다. 아래에 이 함수의 선언을 보여 주었다.

HCURSOR LoadCursor(HINSTANCE hInst, LPCSTR lpszName);

이 함수는 호출이 성공하면 유표의 손잡이를 돌려 주며 실패하면 NULL 을 돌려 준다. hInst는 마우스유표를 포함하는 모듈의 손잡이이며 lpszName에는 유표의 이름을 설정한다. 그러나 Windows 에 미리 갖추어 진 유표를 사용하는 경우에는 첫 파라메터에 NULL 을 설정하고 두번째 파라메터에는 마크로를 리용하여 미리 갖추어 진 유표의 종류를 설정한다. 아래에 Windows 에 미리 갖추어 진 유표종류의 몇가지를 보여 주었다.

유표의 마크로	형 태
IDC_ARROW	표준화살표유표
IDC_CROSS	십자유표
IDC_HAND	손모양유표(Windows 2000 에 추가된것)
IDC_IBEAM	I 자형유표
IDC_WAIT	모래시계유표

골격코드에서는 작성할 창문의 배경색으로 흰색을 지정하며 그 *붓*(Brush)의 손잡이는 GetStockObject( )라는 API 함수에서 얻는다. 붓(Brush)이란 미리 정의된 크기, 색, 무늬로 화면을 색칠하기 위한 자원이다. GetStockObject( )는 붓, 펜, 문자서체 등 여러가지 표준적인 화면객체의 손잡이를 얻는데 쓰인다. 아래에 함수의 선언을 보여 주었다.

HGDIOBJ GetStockObject(int object);

이 함수는 object 로 지정된 객체의 손잡이를 돌려 준다. 함수호출이 실패한 경우에는 NULL 을 돌려 준다. HGDIOBJ 라는 자료형은 GDI 손잡이를 의미한다. 아래에 자주 사용되는 미리 정의된 여러가지 붓을 보여 주었다.

마크로의 이름	배경형태
BLACK_BRUSH	검은색의 붓
DKGRAY_BRUSH	진한 회색의 붓
HOLLOW_BRUSH	가운데 구멍이 있는 붓
LTGRAY_BRUSH	연한 회색의 붓
WHITE_BRUSH	흰색의 붓

붓을 얻으려면 우의 마크로를 GetStockObject( )의 파라메터에 지정한다.

창문클라스설정을 끝내면 *RegisterClassEx( )*라는 API 함수를 사용하여 창문클라스를 Windows 2000에 등록한다. 아래에 이 함수의 선언을 보여 준다.

```
ATOM RegisterClassEx(CONST WNDCLASSEX *lpWClass );
```

이 함수는 창문클라스를 식별하기 위한 값을 돌려 준다. *ATOM*이란 WORD에 별명을 붙인것이다. 매개의 창문클라스마다에 유일한 값이 주어 진다. lpWClass 에는 WNDCLASSEX 구조체의 주소를 지정한다.

창문의 작성

창문클라스를 정의하고 등록이 끝나면 응용프로그람에서 *CreateWindow( )*라는 API 함수를 사용하여 그 클라스의 창문을 작성할수 있다. 아래에 그 선언을 보여 주었다.

```
HWND CreateWindow(
    LPCSTR lpClassName,        // 창문클라스이름
    LPCSTR lpWinName,          // 창문제목
    DWORD dwStyle,             // 창문형태
    int X, int Y,              // 창문의 왼쪽 웃좌표
    int Width, int Height,     // 창문의 크기
    HWND hParent,              // 어미창문의 손잡이
    HMENU hMenu,               // 기본차림표의 손잡이
    HINSTANCE hThisInst        // 실체의 손잡이
    LPVOID lpszAdditional      // 보충정보지시자
);
```

륜곽코드를 보면 알수 있는바와 같이 CreateWindow( )의 파라메터들은 암시값 혹은 NULL로 되여 있다. 례하면 X, Y, Width, Height에는 많은 경우

*CW_USEDEFAULT*를 설정하는데 이때 Windows 2000은 창문의 적당한 크기와 위치를 결정해 준다.

이 룬곽코드와 같이 어미창문이 없는 경우에는 hParent에 NULL을 설정한다. (*HWND_DESKTOP*를 설정해도 같다.) 창문에 기본차림표가 없든가 창문클라스에 의하여 정의되는 기본차림표를 사용하는 경우에는 hMenu를 NULL로 한다. (hMenu에는 다른 사용방법도 있다.) 이 경우와 같이 만일 보충정보가 필요 없으면(많은 경우 필요 없다.) lpszAdditional을 NULL로 한다.(LPVOID형은 void *를 가리킨다. 례를 들면 LPVOID가 void형에 대한 long지시자를 대신하여 쓰인다.)

나머지 4개의 파라메터는 프로그람에서 적당히 설정해야 한다. 우선 lpszClassname에 창문클라스의 이름(창문클라스등록시에 지정한 이름)을 설정한다. lpszWinName에는 창문의 제목으로 되는 문자렬을 설정한다. 이것을 빈 문자렬로 할수도 있지만 일반적으로 창문에는 제목을 준다. 작성할 창문의 형태는 dwStyle에 설정한다.

*WS_OVERLAPPEDWINDOW*라는 마크로는 체계차림표, 경계선, 최소화단추, 최대화단추, 닫기단추를 가진 표준적인 창문을 지정한다. 이 형식의 창문이 가장 일반적이지만 임의의 형태를 지정할수도 있다. 그렇게 하려면 요구하는 형식을 가리키는 마크로들을 OR연산자로 결합하여 지정해야 한다. 아래에 몇가지 창문형식들을 보여 주었다.

창문형식을 지정하는 마크로	창문기능
WS_OVERLAPPED	경계선을 가지는 중첩된 창문
WS_MAXMIZEBOX	최대화단추를 가지는 창문
WS_MINIMIZEDBOX	최소화단추를 가지는 창문
WS_SYSMENU	체계차림표를 가지는 창문
WS_HSCROOL	수평흘림띠를 가지는 창문
WS_VSCROLL	수직흘림띠를 가지는 창문

hThisInst는 Windows 2000에서는 무시된다. 그러나 Windows 95/98에서는 여기에 응용프로그람실체의 손잡이를 설정해야 한다. 따라서 여러가지 환경에서의 호환성을 유지하면서 또한 앞으로 생길수 있는 문제점을 방지하기 위하여 룬곽코드에서처럼 hThisInst에는 실체의 손잡이를 설정해야 한다. 이 책에서 작성하는 모든 프로그람들에서 이 수법을 리용한다.

CreateWindow( )함수는 작성된 창문의 손잡이를 돌려 주며 창문을 작성할수 없으면 NULL을 돌려 준다.

창문이 작성되였다고 하여 곧 화면에 표시되는것은 아니다. 창문을 표시하려면 *ShowWindow( )*라는 API함수를 호출한다. 아래에 선언을 보여 주었다.

```
BOOL ShowWindow( HWND hwnd, int nHow);
```

표시할 창문의 손잡이를 hwnd 에 설정한다. 표시방법을 nHow 에 설정한다. 창문이 처음으로 표시될 때는 nHow 에 nWinMode(WinMain 에 주어 지는 파라메터) 를 지정하여야 한다. nWinMode 에는 프로그람기동시에 창문을 표시하는 방법이 주어 져 있다. 이 함수의 호출에 의해 필요에 따라 창문을 표시하거나 삭제할수 있다. nHow 에 지정할수 있는 주요값을 아래에 보여 주었다.

표시방법을 가리키는 마크로	효 과
SW_HIDE	창문을 표시하지 않는다.
SW_MINIMIZE	창문을 최소화하여 아이콘으로 표시한다.
SW_MAXMIZE	창문을 최대화한다.
SW_RESTORE	창문을 보통 크기로 한다.

ShowWindow( )함수는 창문의 마지막 표시상태를 돌려 준다. 이미 창문이 표시되여 있다면 령이 아닌 값을, 창문이 표시되여 있지 않으면 령을 돌려 준다.

류곽코드에서는 필요 없지만 실제로는 모든 Windows 2000 프로그람에서 필요한 처리로 되기때문에 *UpdateWindow( )*함수의 호출도 진행된다. 이 함수의 호출에 의하여 Windows 2000 은 기본창문의 갱신이 필요하다는것을 알리는 통보문을 응용프로그람에 보내게 된다.(이 통보문에 대해서는 다음 장에서 설명한다.)

통보문순환고리

류곽코드의 마지막부분은 *통보문순환고리*이다. 통보문순환고리는 모든 Windows 2000 응용프로그람에서 사용된다. 통보문순환고리의 역할은 Windows 2000 에서 보내온 통보문을 받아 들이고 그것을 처리하는것이다. 응용프로그람의 실행중에는 통보문의 전송이 계속된다. 이 통보문들은 읽어 들여 처리될 때까지 응용프로그람의 통보문대기렬에 보관되여 있다. 응용프로그람에서는 통보문을 읽어 들일 준비가 될 때마다 *GetMessage( )*라는 API 함수를 호출한다. 아래에 선언을 보여 주었다.

```
BOOL GetMessage(LPMSG msg, HWND hwnd, UNIT min, UNIT max);
```

받은 통보문은 msg 에 보관된다. 모든 통보문은 아래에 보여 주는 *MSG 구조체*로 표시된다.

```
// MSG 구조체
typedef struct tagMSG
```

```
{
    HWND hwnd;            // 통보문을 보내는 창문
    UNIT message;         // 통보문
    WPARAM wParam;        // 통보문에 대한 정보
    LPARAM lParam;        // 통보문에 대한 정보
    DWORD time;           // 통보문을 보내는 시간
    POINT pt;             // 마우스의 x, y 위치
}MSG;
```

MSG 구조체에서는 통보문을 보내는 창문의 손잡이가 hwnd 에 보관된다. 모든 Windows 2000 통보문은 32bit 의 옹근수로서 message 에 보관된다. 매개 통보문과 관련된 정보는 wParam, lParam 에 보관된다. WPARAM 과 LPARAM 도 32bit 이다.

이식과 관련한 요점 : MSG 구조체의 message 성원은 16bit Windows 에서 16bit 로 되여 있다. 이 성원은 Windows 2000 에서는 32bit 이다. 이와 마찬가지로 16bit Windows 에서는 16bit 인 wParam 성원도 Windows 2000 에서는 32bit 이다.

통보문이 발송된 시간은 time 성원에 ㎳단위로 보관된다.

pt 성원에는 통보문이 발송될 때의 마우스자리표가 보관된다. 자리표가 보관되는 *POINT* 구조체는 아래와 같이 정의되여 있다.

```
typedef struct tagPOINT {
   LONG  x,  y;
}  POINT;
```

응용프로그람의 통보문대기렬에 통보문이 존재하지 않는 경우는 GetMessage( )의 호출에 의해 Windows 2000 에 조종이 넘겨진다.

GetMessage( )의 hwnd 파라메터는 통보문을 받아 들이는 창문을 지정하기 위한것이다. 한개의 응용프로그람이 여러개의 창문으로 구성되는 경우도 있으므로 그가운데서 특정한 창문에 보내는 통보문만을 받아 들이게 해야 하는 경우도 있다. 응용프로그람에 전송되여 오는 모든 통보문을 받으려면 이 파라메터에 NULL 을 설정한다.

GetMessage( )의 나머지 두개의 파라메터는 받아 들이는 통보문의 범위를 지정한것이다. 일반적으로는 응용프로그람에서 모든 통보문을 받아 들인다. 그러자면 륜곽코드에서처럼 min 과 max 를 0 으로 설정해야 한다.

사용자가 프로그람을 완료하면 GetMessage( )는 령을 돌려 주며 이것에 의하여 통보문순환고리가 끝난다. 그밖의 경우에는 령 아닌 값을 돌려 준다. 오유가 발생한 경우에는 -1 을 돌려 준다. 그러나 오유가 발생하는 일은 거의 없으므로 이 책의 프로그람들에서는 그에 대응하지 않고 있다.

통보문순환고리내에서는 두개의 함수가 호출된다. 첫 함수는 *TranslateMessage( )*라는 API 함수이다. 이 함수는 Windows 2000 에 의하여 생성된 가상건코드를 문자통보문으로 변환한다.(가상건코드에 대해서는 후에 설명한다.) 모든 응용프로그람에 필수적인 것은 아니지만 프로그람을 건반으로도 조작할수 있게 하기 위해서 대부분의 경우에 TranslateMessage( )가 사용된다.

통보문이 호출되여 변환되면 *DispatchMessage( )* 라는 API 함수를 사용하여 통보문을 Windows 2000에 발송한다. Windows 2000은 프로그람의 창문함수에 넘길 때까지 이 통보문을 보관한다.

통보문순환고리가 끝나면 WinMain( )함수는 Windows 2000 에 msg.wParam 의 값을 돌리여 완료한다. 이 파라메터에는 프로그람을 완료할 때의 완료코드가 보관된다.

창문함수

류곽코드에 들어 있는 두번째 함수는 *창문함수*이다. 여기에서는 WindowFunc( )라는 함수이름으로 되여 있지만 다른 이름으로 할수도 있다. 창문함수에는 Windows 2000 으로부터 통보문이 넘어 온다. MSG 구조체의 첫 4 개의 성분이 이 함수의 호출파라메터들로 된다. 그러나 류곽코드에서 참조되는 파라메터는 통보문을 가리키는 파라메터뿐이다.

류곽코드의 창문함수는 WM_DESTROY 라는 통보문만을 처리한다. 이 통보문은 사용자가 프로그람을 완료할 때 발송된다. 이 통보문을 받은 때는 *PostQuitMessage( )*라는 API 함수를 호출해야 한다. 이 함수의 파라메터에 주어지는 값은 WinMain( )내의 msg.wParam 에 표시되는 완료코드로 된다. PostQuitMessage( )의 호출에 의하여 응용프로그람에는 WM_QUIT 통보문이 전송된다. 이에 의해 GetMessage( )는 false 를 돌려주고 프로그람이 완료되게 된다.

*WindowFunc( )*에서 받는 그밖의 통보문들은 모두 *DefWindowProc( )*를 호출하여 Windows 2000 에 넘기여 암시적처리를 진행한다. 모든 통보문은 어떤 방법으로든 처리해야 하므로 이 수법을 사용할 필요가 있다.

매 통보문을 처리한 다음 창문함수의 돌림값으로서 적당한 값을 돌려 주어야 한다. 대부분의 통보문은 령을 돌려 주면 좋게 되여 있다. 그러나 다른 값을 돌려 주어야 하는 경우도 있다.

다시 한보 전진

창문의 위치

많은 응용프로그람들에서는 프로그람기동시 창문의 위치와 크기의 설정을 Windows 2000 에 맡기는것이 보편적이지만 이것을 자체로 설정할수도 있다. 이를 위하여서는 창문의 왼쪽웃모서리의 자리표, 창문의 너비, 높이를 CreateWindow( )에서 지정해야 한다. 실례로 륜곽코드에서 CreateWindow( )를 호출하는 부분을 아래의 프로그람코드로 변경하면 창문은 화면의 왼쪽웃부분에 너비 300, 높이 100 으로 표시된다.

```
hwnd = CreateWindow(
    szWinName,                  // 창문클라스이름
    "Window 2000 Skeleton",     // 제목
    WS_OVERLAPPEDWINDOW,        // 창문의 형태
    0,                          // x 자리표
    0,                          // y 자리표
    300,                        // 너비
    100,                        // 높이
    NULL,                       // 어미창문이 없음
    NULL,                       // 차림표 없음
    hThisInst,                  // 프로그람실체손잡이
    NULL                        // 보조파라메터함수
);
```

창문의 위치나 크기를 지정하는 경우에는 이것이 *장치단위*로 표시된다는것을 명심해야 한다. 장치단위란 장치에서 사용되는 물리단위(여기에서는 화소)이다. 이것은 자리표와 크기가 화면에 대하여 설정된다는것을 의미한다. 화면의 왼쪽웃모서리의 자리표가 0,0 으로 된다. 후에 설명하지만 창문에 대한 출력은 *론리단위*로 표시된다. 론리단위는 창문에서 사용되는 현재의 넘기기방식에 따라 차이난다. 창문의 위치는 화면전체에 대하여 지정하는것이므로 CreateWindow( )에서는 론리단위가 아니라 물리단위가 사용된다.

모듈정의파일

16bit 의 Windows 에서는 모든 프로그람에서 모듈정의파일이 필요하다.

모듈정의파일은 16bit 의 환경에서 필요되는 어떤 정보나 설정이 기록된 본문파일이다. 그러나 Windows 2000 의 32bit 방식에서는 거의나 사용되지 않는다. 모든 모듈정의파일에는 확장자로서 .DEF 가 사용된다. 실례로 우에서 본 륜곽코드의 모듈정의파일이라면 SKEL.DEF 라는 파일이름을 가지게 된다. Windows 3.1 파의 아래방향호환성을 유지하기 위한 모듈정의파일을 아래에 표시한다.

```
DESCRIPTION 'Skeleton Program'
EXETYPE WINDOWS
CODE PRELOAD MOVEABLE DISCARDABLE
DATA PRELOAD MOVEABLE MULTIPLE
HEAPSIZE 8192
STACKSIZE 8192
EXPORTS WindowFunc
```

모듈정의파일에는 프로그람의 이름이나 설명이 서술되여 있다.(DESCRIPTION) 이것은 생략할수 있다. 실행파일이 DOS 용 등이 아니라 Windows 용이라는것도 표시되여 있다.(EXETYPE) CODE 명령문은 프로그람전체를 기동시에 적재(PRELOAD)한다는것과 코드를 기억기상에서 이동할수 있다는것(MOVEABLE), 필요에 따라 코드를 기억기에서 파괴하거나 재적재할수 있다는것(DISCARDABLE)을 Windows 2000 에 알려 준다.

또한 프로그람의 자료도 실행시에 적재되여 기억기상에서 이동할수 있다는것과 프로그람의 개개의 실체가 여러가지 자체의 자료를 가진다는것(MULTIPLE)을 표시하고 있다. 프로그람의 더미령역의 크기(HEAPSIZE)와 탄창령역의 크기(STACKSIZE)도 지정한다.

마감으로 창문함수의 이름을 수출한다.(EXPORTS) 수출에 의해 Windows 3.1 이 창문함수를 호출하게 된다.

참고 : 모듈정의파일은 Windows 2000 프로그람을 작성할 때는 거의 쓰이지 않는다.

이름붙이기규약

이 장을 마치면서 함수나 변수의 이름을 붙이는 방법을 설명하기로 한다. Windows 프로그람을 작성해본 경험이 없는 사람들은 륜곽코드에서 쓰인 변수나 파라메터의 이름들에서 의문을 가졌을것이다. 이러한 이름들은 Microsoft 가 고안한 Windows 프로그람

작성에서의 *이름붙이기규약*에 따른것이다. 함수의 경우에 동사, 명사의 순서로 함수이름을 달아준다. 동사와 명사의 첫 문자는 대문자로 한다. 이 책의 많은 실례프로그람들에서는 이 이름붙이기규약을 리용하고 있다.

변수의 경우에는 변수의 이름속에 자료형을 반영하는 정보를 삽입하는 약간 복잡한 수법을 고안하였다. 이것은 변수이름의 앞에 소문자로 자료형을 나타내는 앞붙이를 붙여주는것이다. 자료형을 나타내는 앞붙이를 표 2-1에 주었다.

그러나 자료형의 앞붙이를 사용하는 방법에는 여러가지로 고려해야 할 문제가 있다. 대다수의 Windows 프로그람작성자들은 이 방법을 채용하고 있지만 Windows 프로그람작성자가 아닌 사람들은 리용하지 않고 있다.

표 2-1. 변수의 자료형을 표시하는 앞붙이

앞 붙 이	자 료 형
b	론리형(1byte)
c	문자(1byte)
dw	부호 없는 긴 옹근수
f	16bit의 비트마당(기발)
fn	함수
h	손잡이
l	긴 옹근수
lp	long 지시자
n	짧은 옹근수
p	지시자
pt	화면상의 자리표를 가리키는 긴 옹근수
w	부호 없는 짧은 옹근수
sz	령으로 끝나는 문자렬에 대한 지시자
lpsz	령으로 끝나는 문자렬에 대한 long지시자
rgb	RGB색값을 가리키는 긴 옹근수

제 3 장

응용프로그람의 진수: 통보문과 기본입출력

제 2 장에서 작성한 륜곽코드는 Windows 2000 프로그람의 기틀을 보여주지만 그자체가 수행하는 기능은 없다. 실용적인 프로그람으로 되자면 두가지 기본적인 처리를 할수 있어야 한다.

첫째로, 여러가지 통보문에 응답하는것이다. 통보문에 대한 응답은 모든 Windows 2000 응용프로그람의 중심적인 처리로 된다. 둘째로, 프로그람이 어떤 의미를 가지는 정보를 사용자에게 제공하는것이다.(실례로 화면에 정보를 출력하는것)

다른 조작체계와는 달리 Windows 에서는 사용자에게 정보를 표시하여 제공하는것이 간단하지 않다. 사실 출력과 관련된 처리는 모든 Windows 응용프로그람에서 많은 부분을 차지하게 된다. 통보문의 처리와 정보표시기능이 없이는 실용적인 Windows 프로그람을 작성할수 없다.

그러므로 이 장에서는 통보문에 대한 응답과 기본입출력에 중점을 두고 학습한다. 먼저 문자렬을 표시하는 간단한 방법을 시험해 보자.

통 보 칸

화면에 정보를 출력하는 가장 간단한 방법은 *통보칸*을 사용하는것이다. 통보칸은 사용자에게 *통보*를 표시하거나 응답을 기다리게 하기 위한 창문이다. 프로그람작성자자신이 작성해야 하는 다른 류형의 창문들과는 달리 통보칸은 조작체계에서 제공해 준다. 일반적으로 통보칸의 역할은 어떤 사건의 발생을 사용자에게 알려 주는것이다. 그러나 통보내용에 응답하는 어떠한 선택의 여지를 사용자에게 주기 위한 목적에도 통보칸을 사용할수 있다. 실례로 [OK] 나 [CANCEL] 등을 사용자가 선택하게 하는데 통보칸을 사용한다.

참고 : 통보칸이라는 용어에서 통보라는것은 프로그람의 창문함수에 전송되는 Windows 2000 의 통보문이 아니라 화면에 표시되는 문자렬을 의미한다. 기술적으로 중복되는 점도 있지만 통보칸과 통보문은 전혀 다른 내용이다.

*MessageBox( )*라는 API 함수를 사용하여 통보칸을 작성한다. 아래에 이 함수의 선언을 보여 주었다.

```
int MessageBox(HWND hwnd, LPCSTR lpText, LPCSTR lpCaption
               UINT MBType);
```

hwnd 는 어미창문의 손잡이이다. lpText 는 통보칸의 내부에 표시하려는 문자렬에 대한 지시자이다. lpCaption 이 지적하는 문자렬은 통보칸의 제목이다. MBType 값은 통보칸안에 표시하는 단추나 아이콘의 종류 등 통보칸의 형식을 지정한다. 주요한 설정값을 표 3-1 에 주었다. 이 마크로들은 WINDOWS.H 에 정의되여 있다. 상반된 의미를 가지지 않는 마크로들은 OR 연산자로 결합하여 지정할수 있다.

표 3-1. MBType 의 주요설정값

값	효 과
MB_ABORTERETRYIGNORE	[Stop], [Retry], [Ignore] 단추를 표시한다.
MB_CANCELTRYCONTINUE	[Cancel], [Retry], [Continue] (Windows 2000 에 추가된것)를 표시한다.
MB_ICONEXCLAMATION	감탄부호아이콘을 표시한다.

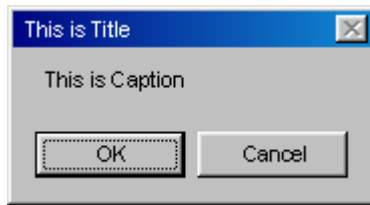
MB_ICONERROR	정지아이콘을 표시한다.
MB_ICONINFORMATION	정보아이콘을 표시한다.
MB_ICONQUESTION	의문표아이콘을 표시한다.
MB_ICONSTOP	MB_ICONERROR 와 같다.
MB_OK	[OK]단추를 표시한다.
MB_OKCANCEL	[OK]와 [Cancel]단추를 표시한다.
MB_RETRYCANCEL	[Retry], [Cancel]단추를 표시한다.
MB_YESNO	[Yes], [No]단추를 표시한다.
MB_YESNOCANCEL	[Yes], [No], [Cancel]단추를 표시한다.

MessageBox( )는 통보칸에 대한 사용자의 응답을 돌려 준다. 돌림값은 아래와 같다.

눌린 단추	돌 림 값
[Stop]	IDABORT
[Retry]	IDRETRY
[Ignore]	IDIGNORE
[Cancel]	IDCANCEL
[Continue]	IDCONTINUE(Windows 2000 에 추가된것)
[No]	IDNO
[Yes]	IDYES
[OK]	IDOK
[Retry]	IDTRYAGAIN(Windows 2000 에 추가된것)

MBType 의 값에 따라 표시되는것은 단추나 아이콘만이라는데 주의해야 한다. 일반적으로 통보칸은 사용자에게 통보문과 [OK]단추를 제공하는데 많이 쓰인다. 이러한 경우에는 프로그람에서 돌림값을 무시한다. MessageBox( )함수를 호출하면 Windows 2000 은 통보칸을 표시해 준다. MessageBox( )는 자동적으로 창문을 작성하고 그 내부에 통보내용을 표시한다. 그밖에는 아무런 처리도 진행하지 않는다. 실례로 아래와 같이 MessageBox( )를 호출하면 다음의 통보칸이 표시된다.

i = MessageBox(hwnd, “This is Caption”, “ This is Title”, MB_OKCANCEL);

사용자가 누른 단추의 종류에 따라 i 값은 *IDOK*, *IDCANCEL* 의 어느 하나로 된다. 통보칸은 례를 들면 오유발생과 같이 어떤 사건이 발생하였다는것을 사용자에게 알려 주는데 많이 쓰인다. 통보칸의 사용방법은 간단하므로 화면에 어떤 오유수정정보를 표시하는데도 편리한 방법으로 쓰인다. 이 책의 많은 실례프로그람들에서 화면에 정보를 표시하기 위한 편리한 수단으로서 통보칸을 사용한다.

Windows 2000 의 새로운 기능 : MessageBox( )의 형태를 표시하는 마크로 MB_CANCELTRYCONTINUE 와 돌림값의 IDTRYAGAIN, IDCONTINUE 는 Windows 2000 에 새롭게 추가된것이다. 번역프로그람의 종류에 따라서 이러한 추가선택을 사용하기 위해서 WINVER 마크로를 0x500 으로 정의할 필요가 있다.

Windows 2000 의 통보문

Windows 2000 에서 통보문은 유일한 32bit 의 옹근수값으로 되여 있다. Windows 2000 은 통보문을 전송하여 프로그람과 련계를 취한다. 매개 통보문들은 어떤 사건에 대응되여 있다. 실례로 사용자가 건반을 눌렀다는것을 표시하는 통보문, 마우스가 이동하였다는것을 표시하는 통보문, 창문의 크기가 변경되였다는것을 표시하는 통보문이 있다. 매개 통보문은 수값으로 식별할수 있지만 일반적으로는 Windows 2000 의 모든 통보문을 정의한 마크로를 사용한다. 즉 통보문을 식별할 때 실제의 수값이 아니라 마크로의 이름을 사용한다. 프로그람에 WINDOWS.H 를 포함시키면 일반적인 통보문들의 마크로가 정의된다. 아래에 Windows 2000 의 일반적인 통보문마크로를 몇개 보여 주었다.

WM_CHAR	WM_PAINT	WM_MOVE
WM_CLOSE	WM_LBUTTONUP	WM_LBUTTONDOWN
WM_COMMAND	WM_HSCROLL	WM_SIZE

통보문과 관련된 두개의 값이 통보문과 함께 전달된다. 첫번째 값의 자료형은 *WPARAM* 이며 다른 값의 자료형은 *LPARAM* 이다. Windows 2000 에서 이 자료형들은 모두 32bit 옹근수값을 나타낸다. 이 값들은 일반적으로 wParam, *lParam* 이라고 한다. *wParam*, *lParam* 의 내용은 통보문의 종류에 따라 다르다. 실례로 마우스의 자리표나 눌리운 건의 값 등이 보관되여 있다. 통보문을 처리할 때 wParam, lParam 의 내용을 참고한다.

이식과 관련한 요점 : 16bit Windows 에서는 wParam 이 16bit 값으로 되여 있다. Windows 2000 에서는 wParam 이 32bit 의 값이다. 16bit 환경과 32bit 환경에서는 통보문의 의미가 달라 지는 경우가 있다. 이러한 차이가 16bit 프로그람을 Windows 2000 에 이식할 때 문제점으로 되는 경우가 있다.

제 2 장에서 설명한바와 같이 프로그람에서 실제로 통보문을 처리하는것은 창문함수이다. 이 함수에는 통보문을 보내는 창문의 손잡이, 통보문자체, wParam, lParam 의 4 개의 파라메터가 주어 져 있다. wParam, lParam 의 내용이 두개 단어의 값으로 분할되여 있는 경우도 있다. 그러므로 Windows 에는 *LOWORD*, *HIWORD* 라는 두개의 마크로가 정의되여 있다. 이 마크로들은 각각 긴 옹근수값의 아래단어와 웃단어를 돌려 준다. 아래에 리용실례를 보여 주었다.

```
x = LOWORD(lParam);
x = HIWORD(lParam);
```

후에 이 마크로들을 실지 사용하게 된다.

Windows 2000 에는 수많은 통보문마크로들이 정의되여 있다. 모든 통보문들을 다 설명할수 없으므로 이 장에서는 중요한것들만 몇가지 설명한다. 기타 마크로들은 필요한 경우에 다음 장들에서 설명한다.

건입력에 대한 응답

Windows 2000 에서 중요한 통보문의 하나는 건반이 눌리워 졌을 때 발생하는 통보문이다. 이 통보문을 *WM_CHAR* 라고 한다. 응용프로그람은 건반으로부터 직접 *건입력*을 받지 못한다. 건반이 눌리워 졌을 때는 능동상태에 있는 창문(현재 입력초점을 가지고 있는 창문)에 통보문이 전송된다. 이 구조를 실제로 확인해 보기 위해 제 2 장에서 작성한 륜곽코드를 확장하여 건입력통보문을 처리해 보자.

WM_CHAR 가 전송되였을 때는 눌리운 건에 해당한 ASCII 코드가 wParam 에 보관되여 있다. LOWORD(lParam)에는 건이 눌리워 진 상태에 있을 때의 건반복회수가 보관되여 있다. HIWORD(lParam)의 매 비트는 표 3-2 에 표시한 의미를 가진다.

표 3-2. HIWORD(lParam)이 가리키는 건반정보

bit	의 미
15	건이 놓여져 있으면 on, 눌리워 져 있으면 off
14	통보문을 전송하기전에 건이 눌리워 져 있으면 on, 눌리워 져 있지 않으면 off
13	[Alt]건이 동시에 눌리워 진 때 on, 눌리워 져 있지 않으면 off
12	예약
11	예약
10	예약
9	예약
8	확장건반의 특수건이 눌리워 져 있으면 on, 그렇지 않으면 off
7-0	제작자에 의존하는 건코드(주사코드)

이제 작성하게 될 프로그람에서는 눌리워 진 건값을 보관하고 있는 wParam만을 리용한다. 그러나 Windows 2000 은 건입력에 관한 체계의 상세정보를 제공해 준다는것을 기억해 두어야 한다. 물론 이 정보를 리용하는가 무시하는가는 구체적인 상황에 따라 결정된다.

WM_CHAR통보문을 처리하자면 프로그람의 창문함수의 switch문에 코드를 추가해야 한다. 실례 3-1 에 *건입력*내용을 통보칸에 표시하는 프로그람을 보여 주었다.

실례 3-1. WM_CHAR 프로그람

```
// WM_CHAR 통보문의 처리

#include <windows.h>
#include <cstring>
#include <cstdio>

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);

char szWinName[] = "MyWin"; // 창문클라스의 이름

char str[255] = ""; // 출력할 문자렬을 보관한다.

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
```

```
{
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;

  // 창문클라스를 정의한다.
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst;      // 실체의 손잡이
  wcl.lpszClassName = szWinName; // 창문클라스의 이름
  wcl.lpfnWndProc = WindowFunc;  // 창문함수
  wcl.style = 0;                 // 암시적형식

  wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
  wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
  wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

  wcl.lpszMenuName = NULL; // 클라스차림표는 없다.
  wcl.cbClsExtra = 0;       // 보조기억기령역은 필요 없다.
  wcl.cbWndExtra = 0;

  // 창문의 배경색을 흰색으로 한다.
  wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

  // 창문클라스를 등록한다.
  if(!RegisterClassEx(&wcl)) return 0;

  /* 창문클라스가 등록되였다면
       창문을 작성할수 있다. */
  hwnd = CreateWindow(
    szWinName, // 창문클라스의 이름
    "Processing WM_CHAR Messages", // 제목
    WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
    CW_USEDEFAULT, // X자리표를 Windows가 결정하게 한다.
    CW_USEDEFAULT, // Y자리표를 Windows가 결정하게 한다.
    CW_USEDEFAULT, // 창문의 너비를 Windows가 결정하게 한다.
    CW_USEDEFAULT, // 창문의 높이를 Windows가 결정하게 한다.
    NULL,          // 어미창문은 없다.
```

```
    NULL,          // 차림표는 없다.
    hThisInst,     // 실체의 손잡이
    NULL           // 추가파라메터는 없다.
  );

  // 창문을 표시한다.
  ShowWindow(hwnd, nWinMode);
  UpdateWindow(hwnd);

  // 통보문순환고리를 작성한다.
  while(GetMessage(&msg, NULL, 0, 0))
  {
    TranslateMessage(&msg); // 건반통보를 변환한다.
    DispatchMessage(&msg);  // Windows 2000에 조종을 넘긴다.
  }

  return msg.wParam;
}

/* 이 함수는 Windows 2000으로부터 호출되여
    통보문대기렬로부터 꺼낸 통보문을 받아 들인다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  switch(message) {
    case WM_CHAR: // 건입력을  처리한다.
      sprintf(str, "Character is %c", (char) wParam);
      MessageBox(hwnd, str, "WM_CHAR Received", MB_OK);
      break;
    case WM_DESTROY: // 프로그람을 완료한다.
      PostQuitMessage(0);
      break;
    default:
      /* 이 switch문에서 지정된것 이외의 통보문은
Windows 2000에 처리를 맡긴다. */
      return DefWindowProc(hwnd, message, wParam, lParam);
  }
```

```
    return 0;
  }
```

이 프로그람의 실행결과를 그림 3-1 에 보여 주었다.

그림 3-1. WM_CHAR 프로그람의 실행결과

WindowFunc( )프로그람코드중에서 다음의 부분에 주목하시오.

```
case WM_CHAR: // 건입력을 처리한다.
     sprintf(str, "Character is %c", (char) wParam);
     MessageBox(hwnd, str, "WM_CHAR RECEIVED", MB_OK);
     break;
```

case 문에 WM_CHAR 통보문의 처리가 추가되였다는것을 알수 있다. 프로그람을 실행하면 건이 눌리울 때마다 WM_CHAR 통보문이 발생되여 WindowFunc( )에 전송된다. WM_CHAR 를 처리하는 case 문에서는 wParam 에 보관된 문자를 sprintf( )를 사용하여 문자렬로 변환하고 그것을 통보칸에 표시한다.

건반통보의 상세

WM_CHAR 는 가장 일반적으로 처리되는 통보문이지만 그밖에도 여러가지 *건반통보*들이 있다. 사실 WM_CHAR 는 프로그람의 통보문순환고리의 *TranslateMessage( )*에 의하여 생성되는 통합적인 통보문으로 되고 있다.

보다 낮은 준위에서 보면 건이 눌리울 때마다 Windows 2000 은 두개의 통보문을 생성시킨다. 건이 눌리우면 *WM_KEYDOWN* 통보문이 발송되고 건을 놓으면 *WM_KEYUP* 통보문이 발송된다. 가능한 경우에는 WM_KEYDOWN 과 WM_KEYUP 통보문의 조합이 TranslateMessage( )에 의하여 WM_CHAR 통보문으로 변환된다.

그러므로 만일 통보문순환고리에 TranslateMessage( )가 포함되여 있지 않으면 프로그람은 WM_CHAR 통보문을 접수할수 없다. 이것을 확인하기 위하여 앞의 프로그람

에서 TranslateMessage( )를 호출하는 부분을 설명문으로 만들어 놓으시오. 이렇게 하면 프로그람은 건입력에 응답하지 못하게 된다.

WM_KEYDOWN, WM_KEYUP 이 문자입력에 거의나 사용되지 않는 리유는 이 통보문들이 제공하는 정보가 가공되지 않은 자료이기때문이다. 실례로 wParam 에는 *ASCII 코드가* 아니라 가상건코드가 들어 있다. TranslateMessage( )의 기능의 하나로서 shift 건상태를 고려하여 가상건코드를 ASCII코드로 변환하는 기능이 있다. TranslateMessage( )에는 건의 자동반복을 자동적으로 처리하는 기능도 있다. *가상건코드*란 장치에 의존하지 않는 건코드이다. 모든 콤퓨터의 건반에는 ASCII 문자모임에 대응되여 있지 않는 건이 여러개 있다. 방향건이나 기능건 등이 그 실례이다.

모든 건에 대응되여 있는것은 가상건코드이다. 모든 가상건코드의 값은 WINUSER.H(프로그람에 WINDOWS.H 를 인용하면 자동적으로 인용된다.)에 마크로로서 정의되여 있다. 이 마크로들은 VK_로 시작하는 이름을 가진다. 아래에 가상건코드들의 실례를 준다.

가상건코드	대응하는 건
VK_DOWN	아래방향건
VK_LEFT	왼쪽방향건
VK_RIGHT	오른쪽방향건
VK_UP	웃방향건
VK_SHIFT	Shift 건
VK_CONTROL	Ctrl 건
VK_ESCAPE	Esc 건
VK_F1 ～ VK_F24	기능건
VK_HOME	Home 건
VK_END	End 건
VK_INSERT	Insert 건
VK_DELETE	Delete 건
VK_PRIOR	Page Up 건
VK_NEXT	Page Down 건
VK_A ～ VK_Z	영문자건
VK_0 ～ VK_9	수자건

*ASCII 코드*에 대응되는 건에 대해서는 TranslateMessage( )가 가상건코드를 ASCII 코드로 변환하고 이것을 WM_CHAR 통보문과 함께 보낸다. 물론 ASCII 코드에 대응되지 않는 건은 변환할수 없다. 따라서 프로그람에서 ASCII 코드에 대응되지 않는 건입력을 처리하려고 한다면 WM_KEYDOWN 또는 WM_KEYUP(또는 둘 다)를 사용해야 한다.

앞의 프로그람을 확장하여 WM_KEYDOWN 과 WM_CHAR 의 두 통보문을 처리할

수 있게 해 보자.(실례 3-2) WM_KEYDOWN 처리에서는 건이 방향건, Shift 건, Ctrl 건중의 어느 건 인가를 나타낸다. 프로그람의 실행결과는 그림 3-2에서 보여 주었다.

실례 3-2. WM_KEYDOWN 프로그람

```
// WM_KEYDOWN과 WM_CHAR 통보문의 처리

#include <windows.h>
#include <cstring>
#include <cstdio>

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);

char szWinName[] = "MyWin"; // 창문클라스의 이름

char str[255] = ""; // 출력할 문자렬을 보관한다.

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;

  // 창문클라스를 정의한다.
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst;     // 실체의 손잡이
  wcl.lpszClassName = szWinName; // 창문클라스의 이름
  wcl.lpfnWndProc = WindowFunc;  // 창문함수
  wcl.style = 0;                 // 체계설정의 형식

  wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
  wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
  wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

  wcl.lpszMenuName = NULL; // 클라스차림표는 없다.
```

```
    wcl.cbClsExtra = 0;       // 보조기억기령역은 필요 없다.
    wcl.cbWndExtra = 0;

    // 창문의 배경색을 흰색으로 한다.
    wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

    // 창문클라스를 등록한다.
    if(!RegisterClassEx(&wcl)) return 0;

    /* 창문클라스가 등록되였으므로
       창문을 작성할수 있다. */
    hwnd = CreateWindow(
      szWinName, // 창문클라스의 이름
      "Processing WM_CHAR and WM_KEYDOWN Messages",
      WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
      CW_USEDEFAULT, // X자리표는 Window가 결정하게 한다.
      CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
      CW_USEDEFAULT, // 창문의 너비는 Windows가 결정하게 한다.
      CW_USEDEFAULT, //창문의 높이는 Windows가 결정하게 한다.
      NULL,          // 어미창문은 없다.
      NULL,          // 차림표는 없다.
      hThisInst,     // 실체의 손잡이
      NULL           // 추가파라메터는 없다.
    );

    // 창문을 표시한다.
    ShowWindow(hwnd, nWinMode);
    UpdateWindow(hwnd);

    // 통보문순환고리를 작성한다.
    while(GetMessage(&msg, NULL, 0, 0))
    {
      TranslateMessage(&msg); // 건반통보를 변환한다.
      DispatchMessage(&msg);  // Windows 2000에 조종을 넘긴다.
    }

    return msg.wParam;
}
```

```
/* 이 함수는 Windows 2000에서 호출되여 통보문대기렬에서
    꺼낸 통보문을 받아 들인다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  switch(message) {
    case WM_CHAR: // 문자를 처리한다.
      sprintf(str, "Character is %c", (char) wParam);
      MessageBox(hwnd, str, "WM_CHAR Received", MB_OK);
      break;
    case WM_KEYDOWN: // 가상건코드를 처리한다.
      switch((char)wParam) {
        case VK_UP:
          strcpy(str, "Up Arrow");
          break;
        case VK_DOWN:
          strcpy(str, "Down Arrow");
          break;
        case VK_LEFT:
          strcpy(str, "Left Arrow");
          break;
        case VK_RIGHT:
          strcpy(str, "Right Arrow");
          break;
        case VK_SHIFT:
          strcpy(str, "Shift");
          break;
        case VK_CONTROL:
          strcpy(str, "Control");
          break;
        default:
          strcpy(str, "Other Key");
      }
      MessageBox(hwnd, str, "WM_KEYDOWN Received", MB_OK);
      break;
    case WM_DESTROY: // 프로그람을 완료한다.
```

```
        PostQuitMessage(0);
        break;
      default:
        /* 이 switch 문에서 지정된것 이외의 통보문은
  Windows 2000 에 처리를 맡긴다. */
        return DefWindowProc(hwnd, message, wParam, lParam);
    }

    return 0;
  }
```

그림 3-2. WM_KEYDOWN 프로그람의 실행결과

여기서 한가지 중요한 내용을 기억해 두어야 한다. 실례로 [X]와 같이 표준적인 ASCII 코드에 해당한 건을 누른 경우에 프로그람은 두개의 통보문 즉 WM_CHAR 과 WM_KEYDOWN 통보문을 받게 된다.

그것은 건을 누를 때마다 WM_KEYDOWN 통보문이 생성되며 (모든 건입력은 WM_KEYDOWN 통보문을 발생한다.) 만일 ASCII 코드에 대응하는 건인 경우에는 TranslateMessage( )에 의하여 WM_CHAR 통보문이 생성되기때문이다.

기타 건반통보

WM_KEYDOWN 이나 WM_KEYUP 이외에도 아래에 표시한 여러개의 *건반통보*가 있다.

통 보 문	설　　명
WM_SYSKEYDOWN	체계건이 눌리워 졌을 때의 가공되지 않은 건반통보. wParam 에는 가상건코드가 넣어 진다.
WM_SYSKEYUP	체계건을 놓은 때의 가공되지 않은 건반통보. wParam 에는 가상건코드가 넣어 진다.
WM_DEADCHAR	현재의 환경에서 건변환이 진행되지 않는 경우에 TranslateMessage( )에 의해 생성되는 통보문. wParam 에는 문자코드가 넣어 진다.

WM_SYSCHAR	체계건이 눌리워 졌을 때 TranslateMessage( )에 의하여 생성되는 통보문. wParam에 문자코드가 넣어 진다.
WM_SYSDEADCHAR	현재 환경에서 체계건의 변환이 진행되지 않은 경우에 TranslateMessage( )에 의하여 생성되는 통보문. wParam에 문자코드가 넣어 진다.

체계건관련통보문은 [Alt]건과 다른 건이 동시에 눌리워 진 때(실례로 [Alt]+[F]) 생성된다. 모든 건반통보에 있어서 lParam의 값은 WM_CHAR통보문의 lParam과 내용이 같다.

대다수의 프로그람들에서는 통보문순환고리에 TranslateMessage( )를 포함시켜 WM_CHAR 통보문을 처리할수 있게 한다. 그외의 건반통보들은 특수한 용도의 경우에만 사용된다. 다른 프로그람작성환경들에서는 저준위의 건반입력처리가 중요한 문제로 취급되였다. 그러나 Windows 에서는 전혀 그렇지 않다. 왜냐하면 Windows 에서는 편집칸이나 목록조종체 등 건입력을 자동적으로 처리하는 조종체가 제공되여 있기때문이다.

창문에 문자렬을 표시하기

통보칸은 정보를 표시하기 위한 가장 간단한 수단이지만 프로그람작성자들의 모든 요구를 다 만족시키지는 못한다. 통보칸을 사용하는것보다 약간 복잡하기는 해도 정보를 표시하기 위한 다른 수단이 있다. 그것은 창문의퇴자구역에 직접 쓰는것이다. Windows 2000 은 *문자렬*과 도형의 두가지 출력을 지원한다. 여기에서는 문자렬을 출력하기 위한 기본적인 방법을 설명한다. 도형출력과 관련해서는 다음 장에서 설명한다.

창문에 문자렬을 출력하는데는 *C/C++의 표준입출력기능*을 사용하지 않는다. 그 리유는 간단하다. C/C++의 표준입출력함수나 연산자는 표준출력장치에 대하여 출력을 진행한다. 이와 반면에 Windows 프로그람에서는 창문에 대하여 출력을 진행한다. 어떻게 문자렬이 창문에 출력되는가를 보기 위해 건입력된 문자를 통보칸이 아니라 프로그람의 창문에 표시하는 실례프로그람을 보자. 이 기능을 실현하는 WindowFunc( )를 아래에 보여 주었다.

```
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
     WPARAM wParam, LPARAM lParam)
{
     HDC hdc;
     static unsigned j=0;
```

```
        switch (message) {
        case WM_CHAR: //건입력을 처리한다.
                hdc = GetDC(hwnd); //장치상황을 얻는다.
                sprintf(str, "%c",(char)wParam);    //문자를 문자렬로 한다.
                TextOut(hdc, j*10,0,str,strlen(str)); // 문자를 출력한다.
                j++;
                ReleaseDC(hwnd, hdc); // 장치상황을 해제한다.
                break;
        case WM_DESTROY: // 프로그람을 완료한다.
                PostQuitMessage(0);
                break;
        default:
                // 이 switch 문에 지정된것 이외의 통보문은
                // Windows 2000 에서 처리된다.
                return DefWindowProc(hwnd, message, wParam, lParam);
        }
        retrurn 0;
    }
```

case 문의 WM_CHAR 통보문처리부분을 보자. 여기서는 프로그람의 창문에 입력된 문자를 그대로 출력한다. C/C++의 표준입출력함수나 연산자를 사용하는 경우에 비하면 이 코드는 간결하지 못하고 복잡한것으로 보일수도 있다. 그 리유는 Windows 에서는 프로그람과 화면을 결합시킨 구조가 필요되기때문이다.

이 구조를 *장치상황*(Device Context)이라고 부르며 *GetDC( )*를 호출하여 얻을수 있다. 현 단계에서는 장치상황에 대한 정확한 정의를 알 필요는 없다. 이에 대해서는 다음 절에서 설명한다. 장치상황을 얻으면 창문에 출력이 진행되도록 되여 있다. 처리마감에 장치상황이 ReleaseDC( )에 의하여 해제된다. 사용이 끝난 장치상황은 프로그람에서 반드시 해제해야 한다. 장치상황의 수는 기억기의 빈용량에 의하여 제한되며 그 수는 어디까지나 유한개이다. 만일 프로그람에서 장치상황을 해제시키지 않으면 결과적으로 장치상황을 사용할수 없게 되며 다음번 GetDC( )호출이 실패한다. 장치상황해제가 실패한 경우에는 *기억기읽기*(Memory leak)가 발생한다.

*GetDC( )*와 *ReleaseDC( )*는 API 함수들이다. 아래에 이 함수들의 선언을 보여 주었다.

```
HDC GetDC(HWND hwnd);
int ReleaseDC(HWND hwnd, HDC hdc);
```

GetDC( )는 hwnd 로 지정된 창문과 관련되는 장치상황을 돌려 준다. *HDC*형은 장

치상황의 손잡이를 가리킨다. 장치상황을 얻을수 없는 경우에 함수의 돌림값은 NULL로 된다.

ReleaseDC( )는 장치상황이 해제되면 TRUE, 그렇지 않으면 FALSE를 돌려 준다. hwnd는 장치상황을 해제하는 창문의 손잡이이다. hdc는 GetDC()를 호출하여 얻은 장치상황의 손잡이이다.

이식과 관련한 요점 : 16bit Windows 에서는 동시에 사용할수 있는 장치상황의 수를 다섯개로 제한하였다. Windows 2000 에서는 그 수가 빈 기억기용량에 의하여 결정된다.

실제로 문자를 출력하는것은 *TextOut( )*라는 API 함수이다. 아래에 선언을 보여 주었다.

```
BOOL TextOut(HDC hdc, int x, int y, LPCSTR lpstr, int nlength);
```

TextOut( )함수는 lpstr 로 지정된 문자렬을 창문의 x, y 로 지정된 좌표에 출력한다.(암시적으로 자리표단위는 화소이다.) 문자렬의 길이를 nlength 에서 지정한다. TextOut( )함수는 처리가 성공하면 령이 아닌 값을 돌려 주며 실패하면 령을 돌려 준다.

WindowFunc( )에서는 WM_CHAR 통보문을 받을 때마다 사용자가 입력한 문자를 *sprintf( )*를 사용하여 한 문자길이의 문자렬로 변환하고 TextOut( )를 사용하여 창문에 출력한다.

첫 문자는 (0, 0)위치에 표시된다. 창문에서 의뢰자구역의 웃왼쪽모서리가 원점으로 된다. 창문자리표는 화면이 아니라 항상 창문에 대하여 설정된다. 따라서 물리적으로 화면상의 그 어디에 창문이 있어도 첫 문자는 창문의 웃왼쪽모서리에 표시된다. 변수 j 는 현재 입력된 문자를 앞문자의 오른쪽에 표시하기 위한 변수이다. 여기서는 두번째 문자는 (10, 0)위치, 세번째 문자는 (20, 0)위치 등의 순서로 표시된다.

Windows 에서는 자동적으로 위치를 갱신하는 유표라는 개념이 지원되지 않는다. 따라서 TextOut( )를 사용할 때마다 문자를 표시하는 적당한 위치를 지정해야 한다. TextOut( )에서는 행바꾸기문자가 주어 져도 행바꾸기를 할수 없다. 이것은 타브에 대해서도 마찬가지이다. 이러한 처리는 작성자자신이 하여야 한다.

설명을 계속하기에 앞서 프로그람코드에서 변수 j를 증가하는 행을 설명문으로 바꾸어놓고 실험을 해 보자. 이렇게 되면 모든 문자가 (0, 0)위치에 표시된다. Windows 는 도형에 기초한 체계이므로 매개 문자의 크기가 다르고 같은 위치에 문자를 덧써도 그전의 문자는 지워 지지 않는다. 례하면 W 다음에 i 라는 문자를 입력하면 W 의 일부가 표시된 상태로 남아 있다. 문자너비가 같지 않으므로 앞의 실례프로그람에서는 입력한 문자들사이 간격이 균일하지 않다는것을 볼수 있다.

이 실례프로그람에서 창문에 문자렬을 출력하는데 사용한 수법은 아무런 연구가 없

이 사용한 수법이다. 실제의 Windows 2000 응용프로그람에서는 이러한 수법을 사용하지 않는다.

Windows 2000 의 모든 API 함수에서는 창문경계선을 벗어나는 출력은 할수 없게 되여 있다. 출력은 자동적으로 자르기되여 경계선을 벗어나지 못하게 된다. 창문의 오른쪽끝에 이르면 그 이후의 문자는 표시되지 않는다는것을 알수 있다.

하나의 문자를 출력하는 프로그람에 TextOut( )를 사용하는것이 이상하게 생각될수도 있지만 Windows 2000 에는 한개 문자를 출력하기 위한 함수가 따로 없다. 그대신에 Windows 2000 은 대화칸, 차림표, 도구띠 등을 사용하여 사용자와 대화하는 수법을 제공한다. 의뢰자구역에 문자렬을 출력하기 위한 함수는 몇개밖에 없다. 출력할 문자렬의 내용을 잘 준비한 다음 그것을 화면상의 어떤 위치에 표시하려면 TextOut( )를 사용하여야 한다.

창문에 건입력내용을 표시하는 완성된 프로그람을 실례 3-3 에 주었다. 그림 3-3 은 프로그람의 실행결과이다.

실례 3-3. TextOut 프로그람

```
// TextOut( )를 사용한 문자렬의 표시

#include <windows.h>
#include <cstring>
#include <cstdio>

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);

char szWinName[] = "MyWin"; // 창문클라스의 이름

char str[255] = ""; // 출력할 문자렬을 보관한다.

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;

  // 창문클라스를 정의한다.
  wcl.cbSize = sizeof(WNDCLASSEX);
```

```
wcl.hInstance = hThisInst;      // 실체의 손잡이
wcl.lpszClassName = szWinName; // 창문클라스의 이름
wcl.lpfnWndProc = WindowFunc;  // 창문함수
wcl.style = 0;                 // 체계설정의 형식

wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

wcl.lpszMenuName = NULL; // 클라스차림표는 없음
wcl.cbClsExtra = 0;       // 보조기억기령역은 필요 없음
wcl.cbWndExtra = 0;

// 창문의 배경색을 흰색으로 한다.
wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

// 창문클라스를 등록한다.
if(!RegisterClassEx(&wcl)) return 0;

/* 창문클라스가 등록되였으므로
   창문을 작성할수 있다. */
hwnd = CreateWindow(
  szWinName, // 창문클라스의 이름
  "Display WM_CHAR Messages Using TextOut", // 형식
  WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
  CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 창문의 너비는 Windows가 결정하게 한다.
  CW_USEDEFAULT, //창문의 높이는 Windows가 결정하게 한다.
  NULL,           // 어미창문은 없다.
  NULL,           // 차림표는 없다.
  hThisInst,      // 실체의 손잡이
  NULL            // 추가파라메터는 없다.
);

// 창문을 표시한다.
ShowWindow(hwnd, nWinMode);
UpdateWindow(hwnd);
```

```
  // 통보문순환고리를 작성한다.
  while(GetMessage(&msg, NULL, 0, 0))
  {
    TranslateMessage(&msg); // 건반통보를 변환한다.
    DispatchMessage(&msg);  // Windows 2000에 조종을 넘긴다.
  }

  return msg.wParam;
}

/* 이 함수는 Windows 2000으로부터 호출되여
   통보문대기렬로부터 꺼낸 통보문을 받아 들인다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  HDC hdc;
  static unsigned j=0;

  switch(message) {
    case WM_CHAR: // 건입력을 처리한다.
      hdc = GetDC(hwnd); // 장치상황을 얻는다.
      sprintf(str, "%c", (char) wParam); // 문자를 문자렬로 한다.
      TextOut(hdc, j*10, 0, str, strlen(str)); // 문자를 출력한다.
      j++; // 이 행을 설명문으로 해보시오
      ReleaseDC(hwnd, hdc); // 장치상황을 얻는다.
      break;
    case WM_DESTROY: // 프로그람을 끝낸다.
      PostQuitMessage(0);
      break;
    default:
      /* 이 switch문에서 지정된것 이외의 통보문은
         Windows 2000에 처리를 맡긴다. */
      return DefWindowProc(hwnd, message, wParam, lParam);
  }

  return 0;
}
```

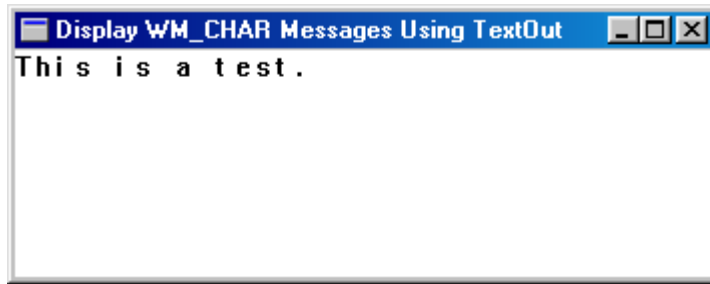



그림 3-3. TextOut 프로그람의 실행결과

장 치 상 황

앞의 프로그람이 보여 주는바와 같이 창문에 출력하려면 먼저 장치상황을 얻어야 한다. 또한 창문함수를 벗어 나기전에 장치상황을 해제하여야 한다.

여기서는 *장치상황*이란 무엇인가를 설명한다. 장치상황이란 창문의 *출력환경*을 표시하는 구조체로서 장치구동기나 현재 서체의 종류 등 다양한 파라메터들을 보관한다. 뒤에서 설명하지만 창문의 출력환경을 조종하는것은 매우 편리한것이다.

응용프로그람에서 창문의 의뢰자구역에 정보를 출력하기전에 장치상황을 얻지 않으면 출력대상으로 되는 창문과 프로그람을 련관시킬수 없다. TextOut( )나 다른 출력함수는 장치상황의 손잡이를 필요로 하므로 이 규칙은 자동적으로 강요되게 된다.

WM_PAINT 통보문의 처리

응용프로그람이 받는 통보문들중에서 가장 중요한것의 하나는 *WM_PAINT* 이다. 이 통보문은 프로그람에서 창문내용을 다시 표시할 필요가 있을 때 보낸다. 이 통보문이 왜 중요한가를 리해하기 위하여 앞절에서 작성한 프로그람을 실행하여 여러개 문자를 입력해 보자.

다음에 창문을 최소화하였다가 다시 원래 크기로 해 보자. 본래의 크기로 복귀된 창문에는 입력된 문자가 표시되여 있지 않다는것을 알수 있다. 창문이 다른 창문밑에 가리웠다가 다시 표시된 경우에도 문자가 표시되지 않는다는것을 알수 있다.

그 원인은 Windows 가 창문내용을 보관하지 않기때문이다. 창문내용을 관리하는것

은 프로그람자체의 역할로 되여 있다. 이것을 프로그람적으로 실현할수 있도록 하기 위해 창문내용을 다시 그릴 필요가 제기될 때마다 WM_PAINT 통보문이 전송되도록 되여 있다.(이 통보문은 창문이 처음으로 창조될 때도 발송된다.) 프로그람에서는 이 통보문을 받을 때마다 창문내용을 다시 그려야 한다.

WM_PAINT 통보문에 응답하는 방법을 설명하기에 앞서 Windows 가 창문내용을 자동적으로 재표시하여 주지 않는 리유를 알아두는것이 좋을것이다. 그 리유는 단 한가지로서 임의의 상황에서 창문의 내용을 잘 알고 있고 쉽게 다시 표시할수 있는것은 조작체계가 아니라 사용자가 작성하는 프로그람이기때문이다.

WM_PAINT 통보문을 처리하자면 먼저 그것을 창문함수의 switch 문에 추가하여야 한다. 앞의 프로그람에 WM_PAINT 를 추가하는 례를 아래에 주었다.

```
case WM_PAINT : // 재표시요구를 처리한다.
     hdc = BeginPaint(hwnd, &paintstruct); //  장치상황을 얻는다.
     TextOut(hdc, 0, 0, strlen(str));
     EndPaint(hwnd, &paintstruct); // 장치상황을 해제한다.
     break;
```

프로그람코드를 상세하게 고찰해 보자. 우선 장치상황을 GetDc( )가 아니라 *BeginPaint( )* 를 사용하여 얻었다는데 주목해야 한다. 여러가지 리유로 하여 WM_PAINT 통보문을 처리할 때는 BeginPaint( )를 사용하여 장치상황을 얻어야 한다. 아래에 선언을 보여 주었다.

```
HDC BeginPaint(HWND hwnd, PAINTSTRUCT *lpPS);
```

BeginPaint( )는 호출에 성공하면 장치상황을 돌려 주고 실패하면 NULL 을 돌려준다. hwnd 는 장치상황을 얻는 창문의 손잡이이다. 두번째 파라메터는 PAINTSTRUCT 구조체의 지시자이다. lpPS 에는 프로그람에서 창문을 재표시하는데 사용되는 정보가 들어 있다. *PAINTSTRUCT* 는 다음과 같이 정의되여 있다.

```
typedef struct tagPAINTSTRUCT {
     HDC hdc;                    // 장치상황의 손잡이
     BOOL fErase;                // 배경을 소거할 필요가 있는 경우 TRUE
     RECT rcPaint;               // 다시 그리는 4각형령역의 자리표
     BOOL fRestore;              // 예약
     BOOL fIncUpdate;            // 예약
     BYTE rgbReserved[32];        // 예약
} PAINTSTRUCT;
```

hdc 에는 재표시할 필요가 있는 창문의 장치상황을 넣는다. 이 장치상황은 BeginPaint( )를 호출하여 돌려 지는 장치상황과 같다. fErase 는 창문배경을 소거할 필요가 있는 경우에 령이 아닌 값으로 된다. 그러나 창문작성시에 배경붓을 지정하면 fErase 를 무시할수 있다. Windows 2000 이 창문을 소거해 주기때문이다.

RECT 는 4 각형령역의 왼쪽웃모서리와 오른쪽아래모서리의 자리표를 지정하기 위한 구조체이다. 아래에 그 내용을 정의한다.

```
typedef tagRECT {
    LONG left, top ;         //왼쪽웃모서리의 자리표
    LONG right, bottom;      //오른쪽아래모서리 자리표
} RECT;
```

PAINTSTRUCT 의 rcPaint 성원은 창문내부에서 다시 그릴 필요가 있는 4 각형령역의 자리표를 보관한다. 여기에서는 창문전체를 다시 그리기하는것을 전제로 하고 있으므로 rcPaint 의 내용을 사용할 필요가 없다. 그러나 실제 프로그람들에서는 이 정보를 리용하는 때도 있다.

장치상황을 얻으면 창문에 대한 출력이 가능하게 된다. 창문의 다시그리기가 완료되면 *EndPaint( )*를 호출하여 장치상황을 해제하여야 한다. 아래에 그 선언을 보여 주었다.

```
BOOL EndPaint(HWND hwnd, CONST PAINTSTRUCT *lpPS);
```

EndPaint( )는 령을 돌려 준다. hwnd 는 다시 그린 창문의 손잡이이다. 두번째 파라메터는 BeginPaint( )를 호출할 때 사용한 PAINTSTRUCT 구조체의 지시자이다.

BeginPaint( )를 사용하여 얻은 장치상황은 반드시 EndPaint( )를 사용하여 해제하여야 한다는것을 잊지 말아야 한다. 또한 BeginPaint( )는 *WM_PAINT* 통보문을 처리할 때만 사용된다는것 역시 명심해 두어야 한다.

WM_PAINT 통보문을 처리하는 완성된 프로그람을 실례 3-4 에 주었다.

실례 3-4. WM_PAINT 프로그람

```
// WM_PAINT 통보문의 처리

#include <windows.h>
#include <cstring>
#include <cstdio>

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
```

```
char szWinName[] = "MyWin"; // 창문클라스의 이름

char str[255] = "Sample Output"; // 출력할 문자렬을 보관한다.

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;

  // 창문클라스를 정의한다.
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst;     // 실체의 손잡이
  wcl.lpszClassName = szWinName; // 창문클라스의 이름
  wcl.lpfnWndProc = WindowFunc;  // 창문함수
  wcl.style = 0;                 // 체계설정의 형식

  wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
  wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
  wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

  wcl.lpszMenuName = NULL; // 클라스차림표는 없다.
  wcl.cbClsExtra = 0;      // 보조기억기령역은 없다.
  wcl.cbWndExtra = 0;

  // 창문의 배경색을 흰색으로 한다.
  wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

  // 창문클라스를 등록한다.
  if(!RegisterClassEx(&wcl)) return 0;

  /* 창문클라스가 등록되였으므로
     창문을 작성할수 있다. */
  hwnd = CreateWindow(
    szWinName, // 창문클라스의 이름
    "Process WM_PAINT Messages", // 제목
```

```
    WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
    CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
    CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
    CW_USEDEFAULT, //창문의 너비는 Windows가 결정하게 한다.
    CW_USEDEFAULT, //창문의 높이는 Windows가 결정하게 한다.
    NULL,          // 어미창문은 없다.
    NULL,          // 차림표는 없다.
    hThisInst,     // 실체의 손잡이
    NULL           // 추가파라메터는 없다.
  );

  // 창문을 표시한다.
  ShowWindow(hwnd, nWinMode);
  UpdateWindow(hwnd);

  // 통보문순환고리를 작성한다.
  while(GetMessage(&msg, NULL, 0, 0))
  {
    TranslateMessage(&msg); // 건반통보를 변환한다.
    DispatchMessage(&msg);  // Windows 2000에 조종을 넘긴다.
  }

  return msg.wParam;
}

/* 이 함수는 Windows 2000으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받아 들인다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  HDC hdc;
  static unsigned j=0;
  PAINTSTRUCT paintstruct;

  switch(message) {
    case WM_CHAR: // 건입력을 처리한다.
      hdc = GetDC(hwnd); // 장치상황을 얻는다.
```

```
        sprintf(str, "%c", (char) wParam); // 문자를 문자렬로 한다.
        TextOut(hdc, j*10, 0, str, strlen(str)); // 문자를 출력한다.
        j++; // 이 행을 설명문으로 해보시오.
        ReleaseDC(hwnd, hdc); // 장치상황을 해제한다.
        break;
      case WM_PAINT: // 다시그리기요구를 처리한다.
        hdc = BeginPaint(hwnd, &paintstruct); // 장치상황을 얻는다.
        TextOut(hdc, 0, 0, str, strlen(str));
        EndPaint(hwnd, &paintstruct); // 장치상황을 해제한다.
        break;
      case WM_DESTROY: // 프로그람을 끝낸다.
        PostQuitMessage(0);
        break;
      default:
        /* 이 switch 문에서 지정된것 이외의 통보문은
           Windows 2000 에 처리를 맡긴다. */
        return DefWindowProc(hwnd, message, wParam, lParam);
    }

    return 0;
  }
```

앞으로 더 나가기에 앞서 이 프로그람을 입력하여 번역한 다음 그것을 실행해 보자. 어떤 문자를 입력하고 창문을 최소화하였다가 다시 원래 크기로 해보자. 창문내용이 다시 표시될 때마다 마감에 입력한 문자만이 자동적으로 다시 표시된다는것을 알수 있다. 그 리유는 변수 str 에 마감에 입력한 문자만이 넣어지도록 되여 있기때문이다.

프로그람을 개작하여 문자렬마감에 문자를 추가하고 WM_PAINT 통보문을 받을 때마다 그 문자렬을 표시하도록 할수도 있다.(다음에 소개하는 실례프로그람에서 구체적인 방법을 보여 준다.)

대역변수 str 는 프로그람의 기동시에 표시되는 “Sample Output”라는 문자렬로 초기화되여 있다. 창문이 작성될 때도 WM_PAINT 통보문이 자동적으로 보내여 지므로 처음에 이 문자렬이 표시된다.

이 프로그람에서 WM_PAINT 통보문을 처리하는 방법은 단순하지만 실지 프로그람에서는 약간 복잡하다. 왜냐면 대부분의 창문에서는 수많은 출력이 필요하기때문이다. 창문의 크기가 변경되였거나 다른 창문밑에 숨겨졌을 때 그 내용을 본래대로 되살리는것은 사용자가 작성하는 프로그람의 역할이므로 이것을 실현하기 위한 어떤 구조를 준비하여야 한다. 실지 프로그람에서는 이것을 다음과 같은 세가지 방법으로 실현할수 있다.

첫번째방법은 다시 표시해야 할 내용을 계산하여 얻는것이다. 이 방법은 사용자가 입력을 하지 않는 프로그람인 경우에 가장 적합하다.

두번째 방법은 어떤 상황에 있어서 사건들을 기록하여 놓았다가 창문을 다시 그릴 필요가 있을 때 그것을 재차 실행하는것이다.

세번째 방법은 가상창문을 작성하여 다시 그릴 필요가 있을 때 그 내용을 실제 창문에 복사하는것이다. 세번째 방법이 가장 일반적인 방법이다.(이에 대해서는 뒤에서 설명한다.) 그러나 어느 방법이 최량인가 하는것은 응용프로그람의 목적에 의해 결정된다.

이 책에 보여 주는 대부분의 실례프로그람들은 창문의 다시그리기를 고려하지 않았다. 그것은 다시그리기를 고려하자면 많은 프로그람코드를 서술해야 하며 실례프로그람에서 설명하려고 하는 요점이 불명확해지기때문이다. 그러나 실제적인 응용프로그람들에서는 Windows 2000 의 류의에 따라 창문을 다시 그릴 필요가 있다.

WM_PAINT 통보문의 생성

프로그람에서 WM_PAINT 통보문을 생성시키는것도 가능하다. 처음에는 왜 프로그람에서 WM_PAINT 통보문을 생성할 필요가 있을가 하고 의문을 가질수 있다. 또한 프로그람이 언제든지 창문을 다시 그릴수 있다고 생각하면 그것은 옳지 않다. 창문을 다시 그리는것은 시간이 걸리는 처리라는것을 명심하여야 한다.

Windows 는 다중과제체계이며 다른 프로그람이 CPU 시간을 요청할수도 있으므로 프로그람측에서 Windows 에 정보를 출력하려고 한다는것을 전달하려는 경우에도 출력시점의 결정은 Windows 에 맡기는것이 최선의 수법이다. 이 수법을 사용하면 Windows 는 체계의 모든 과제에 대하여 CPU 시간을 효률적으로 할당할수 있다.

이 수법을 사용하기 위하여서는 WM_PAINT 통보문을 보내올 때까지 프로그람에서 모든 출력내용을 보관해 두어야 한다. 앞에서 본 실례프로그람에서는 창문의 크기가 변경되였거나 다른 창문밑에 가리웠다가 재표시될 때만 WM_PAINT 를 받도록 되여 있었다. WM_PAINT 통보문이 전송되여 올 때까지 모든 출력내용을 보관해 두고 통보문을 받은 시점에서 사용자와 입출력을 진행하는 경우에는 어떤 출력내용이 대기상태에 있을 때 창문에 WM_PAINT 통보문을 보낼 필요가 있다는것을 Windows 에 알려 주는 어떤 방법이 있어야 할것이다.

Windows 2000 은 이 요구에 따르는 기능을 제공하고 있다. 그러므로 프로그람이 정보를 출력하려는 시점에서 Windows 의 형편에 맞추어 WM_PAINT 통보문을 보낼것을 요구할수 있다.

Windows 가 WM_PAINT 통보문을 발송하도록 하기 위해서 프로그람에서 *InvalidateRect( )*라는 API 함수를 호출한다. 아래에 선언을 보여 주었다.

BOOL InvalidateRect(HWND hwnd, CONST RECT *lpRect, BOOL bErase);

hwnd는 WM_PAINT 통보문을 전송할 대상으로 되는 창문의 손잡이이다. RECT구조체인 lpRect 에는 창문내부의 다시 그리려는 령역을 지정한다. 이 파라메터가 NULL인 경우에는 창문전체가 지정되는것으로 된다. bErase 가 TRUE 인 경우 배경이 소거되고 령인 경우에는 배경이 소거되지 않는다. 호출이 성공한 경우에는 돌림값은 령이 아닌 값으로 되며 성공하지 못한 경우에는 령으로 된다. InvalidateRect( )를 호출하면 창문내용이 무효로 되며 다시 그릴 필요가 있다는것이 Windows 에 통지된다. 이렇게 되여 Windows 가 프로그람의 창문함수에 WM_PAINT 통보문을 보내게 된다.

모든 출력처리를 WM_PAINT 통보문처리에서 하도록 앞의 프로그람을 개조한 프로그람을 실례 3-5 에 주었다. WM_CHAR 통보문에 응답하는 프로그람코드는 매개 문자를 보관해 두고 InvalidateRect( )를 호출하기만 하도록 되여 있다. 이 프로그람에서는 WM_CHAR에서 입력된 매개 문자를 str라는 문자렬마감에 추가한다. 이렇게 하여 창문이 다시 표시될 때마다 마감에 입력된 문자만이 표시된 앞의 프로그람과는 달리 입력된 모든 문자를 포함한 문자렬전체가 출력된다.

실례 3-5. WM_PAINT _SKEL 프로그람

```
/* WM_PAINT 통보문에서 출력을 진행하는
   Windows 의 륜곽코드 */

#include <windows.h>
#include <cstring>
#include <cstdio>

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);

char szWinName[] = "MyWin"; // 창문클라스의 이름

char str[255] = ""; // 출력하려는 문자렬을 보관한다.

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;
```

```
// 창문클라스를 정의한다.
wcl.cbSize = sizeof(WNDCLASSEX);

wcl.hInstance = hThisInst;     // 실체의 손잡이
wcl.lpszClassName = szWinName; // 창문클라스의 이름
wcl.lpfnWndProc = WindowFunc;  // 창문함수
wcl.style = 0;                 // 체계설정의 형식

wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

wcl.lpszMenuName = NULL; // 클라스차림표는 없다.
wcl.cbClsExtra = 0;      // 보조기억기령역은 필요 없다.
wcl.cbWndExtra = 0;

// 창문의 배경색을 흰색으로 한다.
wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

// 창문클라스를 등록한다.
if(!RegisterClassEx(&wcl)) return 0;

/* 창문이 등록되였으므로
    창문을 작성할수 있다. */
hwnd = CreateWindow(
  szWinName, // 창문클라스의 이름
  "Routing Output Through WM_PAINT", // 제목
  WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
  CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 창문의 너비는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 창문의 높이는 Windows가 결정하게 한다.
  NULL,          // 어미창문은 없다.
  NULL,          // 차림표는 없다.
  hThisInst,     // 실체의 손잡이
  NULL           // 추가파라메터는 없다.
);
```

```
    // 창문을 표시한다.
    ShowWindow(hwnd, nWinMode);
    UpdateWindow(hwnd);

    // 통보문순환고리를 작성한다.
    while(GetMessage(&msg, NULL, 0, 0))
    {
      TranslateMessage(&msg); // 건반통보를 변환한다.
      DispatchMessage(&msg);  // Windows 2000 에 조종을 넘긴다.
    }

    return msg.wParam;
}

/* 이 함수는 Windows 2000 으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받아 들인다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message, WPARAM wParam,
                    LPARAM lParam)
{
    HDC hdc;
    PAINTSTRUCT paintstruct;
    char temp[2];

    switch(message) {
      case WM_CHAR: // 건입력을 처리한다.
        sprintf(temp, "%c", (char) wParam); // 문자를 문자렬로 한다.
        strcat(str, temp); // 문자렬에 문자를 추가한다.
        InvalidateRect(hwnd, NULL, 1); // 다시그리기를 진행한다.
        break;
      case WM_PAINT: // 다시그리기요구를 처리한다.
        hdc = BeginPaint(hwnd, &paintstruct); // 장치상황을 얻는다.

        TextOut(hdc, 0, 0, str, strlen(str)); // 문자렬을 출력한다.

        EndPaint(hwnd, &paintstruct); // 장치상황을 해제한다.
        break;
      case WM_DESTROY: // 프로그람을 완료한다.
```

```
        PostQuitMessage(0);
        break;
      default:
        /* 이 switch 문에서 지정된것 이외의 통보문은
           Windows 2000 에 처리를 맡긴다. */
        return DefWindowProc(hwnd, message, wParam, lParam);
    }

    return 0;
}
```

많은 응용프로그람들에서는 의뢰자구역에로의 모든(또는 대부분의) 출력을 WM_PAINT 통보문에서 진행한다. 그 리유는 이미 설명한것과 같다. 그러나 필요한 때 임의의 시점에서 문자렬이나 도형을 출력할수도 있다. 어떠한 방법을 사용하는가는 프로그람의 목적에 의해 결정된다.

다시 한보 전진

통보문의 생성

*InvalidateRect( )*가 WM_PAINT 통보문을 생성하는데 리용된다는것을 앞에서 보았다. 그러면 프로그람에서 일반적인 통보문을 생성하려면 어떻게 하면 좋겠는가 하는 새로운 의문이 생긴다. 그에 대한 대답은 간단하다. SendMessage( )라는 API 함수를 사용하는것이 좋다. SendMessage( )함수를 사용하면 임의의 창문에 통보문을 보낼수 있다. 아래에 선언을 보여 주었다.

```
LRESULT SendMessage(HWND hwnd, UINT message,
                    WPARAM wParam, LPARAM lParam);
```

hwnd 는 통보문을 보내려는 창문의 손잡이이다. 통보문의 값은 message 에 지정한다. 통보문에 부가되는 정보는 wParam, lParam 에 넣는다. SendMessage( )는 통보문에 대한 응답을 돌려 준다.

SendMessage( )는 Windows 2000 의 조종체(특히 공통조종체)를 조작하는데 자주 리용된다. 실제 사용법은 다음 장에서 나오는 여러 실례프로그람들에서 볼수 있다.

물론 통보문을 보내려고 하는 모든 경우에 다 리용하여도 관계 없다. 례하면 아래에 표시한 WindowFunc( )에서는 사용자가 [Ctrl]+[A]건을 누를 때마다 련속적으로 WM_CHAR 통보문을 생성하기 위하여 SendMessage( )를 사용한다.

```
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                  WPARAM wParam, LPARAM lParam);
{
    HDC hdc;
    unsigned i;
    static unsigned j = 0;
    char s[ ] = "Hello There";
    switch(message) {
        case WM_CHAR: // 건입력을 처리한다.
            hdc = GetDC(hwnd); // 장치상황을 얻는다.
            // 사용자가 [Ctrl]+[A]건을 누르면 "Hello There"라고 표시.
             if ((char) wParam == (char) 1) for( i=0; s[i]; i++)
                SendMessage(hwnd, WM_CHAR, (WPARAM) s[i],
                         (LPARAM) 0);
            else {
                sprintf(str, "%c", (char) wParam); // 문자렬을 작성
                TextOut(hdc, j*10, 0, str, strlen(str)); // 문자를 출력
                j++;
            }
            ReleaseDC(hwnd, hdc); // 장치상황을 해제
            break;
        case WM_DESTROY: // 프로그람을 완료
            PostQuitMessage(0);
            break;
        default:
            //이 switch 문에 지정된것 이외의 통보문은 Windows 2000 에서
            // 처리된다.
            return DefWindowProc(hwnd, message, wParam, lParam);
    }
    return 0;
}
```

지금까지 본 프로그람들의 어느 하나의 창문함수를 이 WindowFunc( )로 치환하면 [Ctrl]+[A]건이 눌리울 때마다"Hello There"라는 문자렬이 련속적인 WM_CHAR 통보문으로 되여 프로그람에 전송된다. 이 수법은 프로그람에 건반마크로를 제공하는데 사용할수 있다.

> 다른 한가지 주목해야 할것은 WM_CHAR 통보문이 처리될 때 프로그람이 lParam 을 참고하지 않으므로 SendMessage( )호출에서는 이 파라메터를 간단히 0 으로 하고 있다는것이다. 그러나 다른 파라메터들의 내용은 정보를 전달하기 위한 적절한 값으로 할 필요가 있다.

마우스통보문에 대한 응답

Windows 2000 은 마우스를 기본입력도구로 하는 조작체계이므로 모든 Windows 2000 프로그람은 마우스입력에 대응해야 한다. *마우스통보문*에는 여러가지 종류가 있다. 이 장에서 취급하는 통보문들은 다음과 같다.

WM_LBUTTONDOWN	WM_LBUTTONUP	WM_LBUTTONDBLCLK
WM_RBUTTONDOWN	WM_RBUTTONUP	WM_RBUTTONDBLCLK

대다수 콤퓨터들은 두개의 단추를 가진 마우스를 사용하고 있지만 Windows 2000 에서는 3 개의 단추를 가진 마우스까지도 취급한다. 매 단추를 왼쪽 단추, 가운데단추, 오른쪽 단추라고 한다. Windows 2000 에서는 응용프로그람에서 특정한 지령을 실행하기 위한 특수한 X 단추도 지원하고 있지만 여기에서는 설명하지 않는다. 이 장의 나머지 부분에서는 왼쪽 단추와 오른쪽 단추만을 취급한다.

우선 가장 일반적인 마우스통보문들인 WM_LBUTTONDOWN 과 WM_RBUTTONDOWN 을 설명하자.

이 통보문들은 각각 왼쪽 단추와 오른쪽 단추가 눌리워 졌을 때 생성된다. *WM_LBUTTONDOWN* 혹은 *WM_RBUTTONDOWN*이 발송되는 경우 마우스의 현재 X, Y 위치가 lParam 의 웃단어와 아래단어에 각각 보관된다. wParam 의 값은 다음 절에서 설명하는 다양한 상태정보를 보관한다.

마우스통보문을 받은 때 lParam 에서 X, Y 자리표를 얻는 방법에는 두가지가 있다. 첫번째 방법은 *HIWORD( )*, *LOWORD( )*를 사용하는 보편적인 방법으로서 이 책에서 리용한다. 두번째방법은 *GET_X_LPARAM( )*, *GET_Y_LPARAM( )*이라는 마크로를 사용하는 방법이다. 이 마크로들은 기본적으로 HIWORD, LOWORD 에 별명을 붙인것으로서 기억하기 쉽다. 아래에 선언을 보여 주었다.

```
int GET_X_LPARAM( LPARAM lParam);
int GET_Y_LPARAM( LPARAM lParam);
```

GET_X_LPARAM( ), GET_Y_LPARAM( )를 사용하려면 프로그람에

WINDOWS.H 를 인용하여야 한다. 이 책의 실례프로그람에서는 마우스통보문으로부터 자리표를 얻는데 *HIWORD( )*, *LOWORD( )*를 사용한다.

실례 3-6 은 마우스통보문에 대한 응답을 보여 주는 프로그람이다. *마우스유표*가 프로그람창문내부에 있을 때 마우스단추를 누르면 마우스의 현재위치가 표시된다.

실례 3-6. MouseMsg 프로그람

```
// 마우스통보문의 처리

#include <windows.h>
#include <cstring>
#include <cstdio>

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);

char szWinName[] = "MyWin"; // 창문클라스의 이름

char str[255] = ""; // 출력할 문자렬을 보관한다.

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;

  // 창문클라스를 정의한다.
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst;      // 실체의 손잡이
  wcl.lpszClassName = szWinName; // 창문클라스의 이름
  wcl.lpfnWndProc = WindowFunc;  // 창문함수
  wcl.style = 0;                 // 체계설정형식

  wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
  wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
  wcl.hCursor = LoadCursor(NULL, IDC_ARROW); // 유표의 형식
```

```
    wcl.lpszMenuName = NULL; // 클라스차림표는 없다.
    wcl.cbClsExtra = 0;       // 보조기억기령역은 필요 없다.
    wcl.cbWndExtra = 0;

    // 창문의 배경색을 흰색으로 한다.
    wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

    // 창문클라스를 등록한다.
    if(!RegisterClassEx(&wcl)) return 0;

    /* 창문클라스가 등록되였으므로
  창문을 작성할수 있다. */
    hwnd = CreateWindow(
      szWinName, // 창문클라스의 이름
      "Processing Mouse Messages", // 제목
      WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
      CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
      CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
      CW_USEDEFAULT, // 창문의 너비는 Windows가 결정하게 한다.
      CW_USEDEFAULT, // 창문의 높이는 Windows가 결정하게 한다.
      NULL,          // 어미창문은 없다.
      NULL,          // 차림표는 없다.
      hThisInst,     // 실체의 손잡이
      NULL           // 추가파라메터는 없다.
    );

    // 창문을 표시한다.
    ShowWindow(hwnd, nWinMode);
    UpdateWindow(hwnd);

    // 통보문순환고리를 작성한다.
    while(GetMessage(&msg, NULL, 0, 0))
    {
      TranslateMessage(&msg); // 건반통보를 변환한다.
      DispatchMessage(&msg);  // Windows 2000에 조종을 넘긴다.
    }
```

```
  return msg.wParam;
}

/* 이 함수는 Windows 2000 으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받는다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  HDC hdc;

  switch(message) {
    case WM_RBUTTONDOWN: // 오른쪽 단추를 처리한다.
      hdc = GetDC(hwnd); // 장치상황을 얻는다.
      sprintf(str, "Right button is down at %d, %d",
              LOWORD(lParam), HIWORD(lParam));
      TextOut(hdc, LOWORD(lParam), HIWORD(lParam),
              str, strlen(str));
      ReleaseDC(hwnd, hdc); // 장치상황을 해제한다.
      break;
    case WM_LBUTTONDOWN: // 왼쪽 단추를 처리한다.
      hdc = GetDC(hwnd); // 장치상황을 얻는다.
      sprintf(str, "Left button is down at %d, %d",
              LOWORD(lParam), HIWORD(lParam));
      TextOut(hdc, LOWORD(lParam), HIWORD(lParam),
              str, strlen(str));
      ReleaseDC(hwnd, hdc); // 장치상황을 해제한다.
      break;
    case WM_DESTROY: // 프로그람을 끝낸다.
      PostQuitMessage(0);
      break;
    default:
      /* 이 switch 문에서 지정된것 이외의 통보문은
         Windows 2000 이 처리하게 한다. */
      return DefWindowProc(hwnd, message, wParam, lParam);
  }

  return 0;
}
```

그림 3-4는 프로그람의 실행결과를 보여 준다.

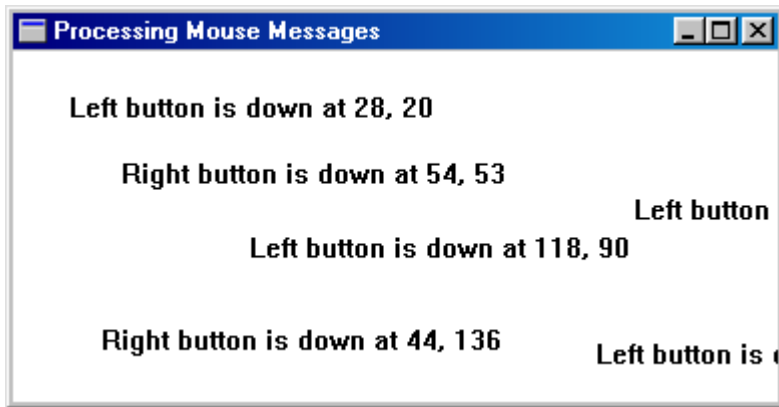



그림 3-4. 마우스통보문프로그람의 실행결과

마우스통보문의 상세

이 장에서 설명하는 모든 마우스통보문에서는 lPara 과 wParam 에 같은 정보가 보관되여 있다. 이미 설명한바와 같이 lParam 에는 통보문이 생성된 때의 마우스자리표가 보관되여 있다. wParam에는 마우스, 건반상태와 관련한 정보가 보관되여 있다. 이것은 아래에 보여 준 값들의 조합으로 된다.

MK_CONTROL
MK_SHIFT
MK_MBUTTON
MK_RBUTTON
MK_LBUTTON
MK_XBUTTON1
MK_XBUTTON2

마우스단추와 동시에 [Ctrl]단추가 눌리워 져 있는 경우에는 wParam 에 *MK_CONTROL* 이 보관된다. 마우스단추와 동시에 [Shift]단추가 눌리워 져 있는 경우에는 wParam에 *MK_SHIFT*가 보관된다.

또한 왼쪽 단추가 눌리워 졌을 때 동시에 오른쪽 단추가 눌리워 져 있는 경우에는 wParam에 *MK_RBUTTON*이 보관된다. 오른쪽 단추가 눌리워 졌을 때 동시에 왼쪽 단추가 눌리워 져 있는 경우에는 wParam에 *MK_LBUTTON*이 보관된다. 왼쪽 혹은 오른

쪽 단추가 눌리워 졌을 때 동시에 가운데단추가 눌리워 져 있는 경우에는 wParam 에 *MK_MBUTTON*이 보관된다.

어떤 X 단추가 눌리워 진 경우에는 *MK_XBUTTON1* 혹은 *MK_XBUTTON2*이 보관된다.

Windows 2000 의 새로운 기능 : MK_XBUTTON1 과 MK_XBUTTON2 는 Windows 2000 에 새로 추가된것이다.

단추놓음통보문의 사용

*마우스단추*가 한번 찰칵되면 프로그람은 두개의 통보문을 받는다. 하나는 단추가 눌리워 졌다는것을 가리키는 WM_LBUTTONDOWN과 같은 *단추누름통보문*이고 다른 하나는 누른 단추를 놓았다는것을 가리키는 *단추놓음통보문*이다. 왼쪽 단추와 오른쪽 단추 놓음통보문은 각각 *WM_LBUTTONUP*과 *WM_RBUTTONUP*이다.

항목을 선택하는것과 같은 조작을 수행하는 응용프로그람들에서는 단추누름통보문보다 단추놓음통보문을 사용하는것이 좋다. 그것은 단추를 누른후에도 사용자가 선택을 취소할수 있기때문이다.

두번 찰칵에 대한 응답

한번찰칵에 응답하는것은 간단하지만 *두번 찰칵*을 취급하는것은 약간 복잡하다. 우선 프로그람에서 두번 찰칵을 받는 설정을 해야 한다. 암시적으로는 프로그람에 두번 찰칵통보문이 전송되지 않는다. 다음 두번 찰칵통보문에 응답하는 프로그람코드를 추가해야 한다.

프로그람이 두번 찰칵통보문을 받도록 하려면 창문클라스를 등록하기전에 WNDCLASSEX 구조체의 Style 성원에 *CS_DBLCLKS*를 지정하는것이 필요하다. 이를 위해 아래와 같은 프로그람코드를 서술한다.

```
wcl.style = CS_DBLCLKS;      // 두번 찰칵을 허가한다.
```

두번 찰칵을 가능하게 하면 프로그람이 *WM_LBUTTONDBLCLIK* 나 *WM_RBUTTONDBLCLIK* 등의 *두번 찰칵통보문*을 받을수 있게 된다. lParam 과 wParam 의 내용은 다른 마우스통보문들과 같다.

*두번 찰칵*이란 짧은 순간에 마우스단추를 두번 반복하여 누르는것이다. 마우스단추를 두번 찰칵하여 두번 찰칵통보문을 발생시키기 위한 *시간간격*을 얻을수도 있고 설정할수도 있다. 두번 찰칵을 위한 시간간격을 얻으려면 *GetDoubleClickTime( )*라는 API 함수를 사용한다. 아래에 선언을 보여 주었다.

```
UINT GetDoubleClickTime(void);
```

이 함수는 두번 찰칵을 발생시키기 위한 시간간격의 웃한계를 ㎳단위로 돌려 준다. 두번 찰칵의 시간간격을 설정하는데는 *SetDoubleClickTime( )*을 사용한다. 아래에 선언을 보여 주었다.

```
UINT SetDoubleClickTime(UINT interval);
```

interval에는 두번 찰칵을 발생시키기 위한 시간간격의 웃한계를 ㎳단위로 설정한다. 0을 설정하면 체계설정의 시간간격이 사용된다.(체계설정의 시간간격은 약 0.5s 이다.) 함수의 호출이 성공하면 0이 아닌 값을 돌려 주며 실패하면 0이 돌려 진다. 실례 3-7의 프로그람은 두번 찰칵통보문에 응답하는 프로그람이다. 이 프로그람은 GetDoubleClickTime( ) 및 SetDoubleClickTime( ) 의 응용실례로도 된다.

웃방향화살건을 누르면 두번 찰칵의 시간간격이 증가한다. 아래방향화살건을 누르면 시간간격은 감소한다. 마우스의 오른쪽 단추 또는 왼쪽 단추중의 어느 하나를 두번 찰칵하면 현재의 두번 찰칵시간간격을 표시하는 통보칸이 표시된다.

두번 찰칵시간간격은 체계전체에 적용되는 설정이므로 값의 설정이 체계의 모든 프로그람들에 영향을 미친다. 이로부터 프로그람의 기동시에 현재의 두번 찰칵시간간격을 보관해 둔다. 일반적으로 응용프로그람들에서 동작과정에 체계의 설정을 변경하였다면 프로그람의 완료시에 원래의 상태로 복귀해 놓아야 한다.

실례 3-7. DblClick 프로그람

```
// 두번찰칵에 대한 응답 및 두번 찰칵시간간격의 조작

#include <windows.h>
#include <cstring>
#include <cstdio>

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);

char szWinName[] = "MyWin"; // 창문클라스의 이름

char str[255] = ""; // 표시할 문자렬을 보관한다.

UINT OrgDblClkTime; // 본래의 두번 찰칵시간간격을 보관한다.

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
```

```
{
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;

  // 창문클라스를 정의한다.
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst;      // 실체의 손잡이
  wcl.lpszClassName = szWinName; // 창문클라스의 이름
  wcl.lpfnWndProc = WindowFunc;  // 창문함수
  wcl.style = CS_DBLCLKS;          // 두번 찰칵을 허가한다.

  wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
  wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
  wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

  wcl.lpszMenuName = NULL; // 클라스차림표는 없다.
  wcl.cbClsExtra = 0;       // 보조기억기령역은 필요 없다.
  wcl.cbWndExtra = 0;

  // 창문의 배경색은 흰색으로 한다.
  wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

  // 창문클라스를 등록한다.
  if(!RegisterClassEx(&wcl)) return 0;

  /* 창문클라스가 등록되였으므로
     창문을 작성할수 있다. */
  hwnd = CreateWindow(
    szWinName, // 창문클라스의 이름
    "Processing Double-Clicks", // 제목
    WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
    CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
    CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
    CW_USEDEFAULT, // 창문의 너비는 Windows가 결정하게 한다.
    CW_USEDEFAULT, // 창문의 높이는 Windows가 결정하게 한다.
    NULL,          // 어미창문은 없다.
```

```
    NULL,          // 차림표는 없다.
    hThisInst,     // 실체의 손잡이
    NULL           // 추가파라메터는 없다.
  );

  // 현재의 두번 찰칵시간간격을 보관한다.
  OrgDblClkTime = GetDoubleClickTime( );

  // 창문을 표시한다.
  ShowWindow(hwnd, nWinMode);
  UpdateWindow(hwnd);

  // 통보문순환고리를 작성한다.
  while(GetMessage(&msg, NULL, 0, 0))
  {
    TranslateMessage(&msg); // 건반통보를 변환한다.
    DispatchMessage(&msg);  // Windows 2000 에 조종을 넘긴다.
  }

  return msg.wParam;
}

/* 이 함수는 Windows 2000 으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받는다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  HDC hdc;
  UINT interval;

  switch(message) {
    case WM_KEYDOWN:
      if((char)wParam == VK_UP) { // 시간간격을 증가시킨다.
        interval = GetDoubleClickTime( );
        interval += 100;
        SetDoubleClickTime(interval);
      }
```

```
    if((char)wParam == VK_DOWN) { // 시간간격을 감소시킨다.
      interval = GetDoubleClickTime( );
      interval -= 100;
      if(interval < 0) interval = 0;
      SetDoubleClickTime(interval);
    }
    sprintf(str, "New interval is %u milliseconds",
            interval);
    MessageBox(hwnd, str, "Setting Double-Click Interval",
               MB_OK);
    break;
  case WM_RBUTTONDOWN: // 오른쪽 단추를 처리한다.
    hdc = GetDC(hwnd); // 장치상황을 얻는다.
    sprintf(str, "Right button is down at %d, %d",
            LOWORD(lParam), HIWORD(lParam));
    TextOut(hdc, LOWORD(lParam), HIWORD(lParam),
           str, strlen(str));
    ReleaseDC(hwnd, hdc); // 장치상황을 해제한다.
    break;
  case WM_LBUTTONDOWN: // 왼쪽 단추를 처리한다.
    hdc = GetDC(hwnd); // 장치상황을 얻는다.
    sprintf(str, "Left button is down at %d, %d",
            LOWORD(lParam), HIWORD(lParam));
    TextOut(hdc, LOWORD(lParam), HIWORD(lParam),
            str, strlen(str));
    ReleaseDC(hwnd, hdc); // 장치상황을 해제한다.
    break;
  case WM_LBUTTONDBLCLK: // 왼쪽 단추의 두번 찰칵을 처리한다.
    interval = GetDoubleClickTime( );
    sprintf(str, "Left Button\nInterval is %u milliseconds",
            interval);
    MessageBox(hwnd, str, "Double Click", MB_OK);
    break;
  case WM_RBUTTONDBLCLK: // 오른쪽 단추의 두번 찰칵을 처리한다.
    interval = GetDoubleClickTime( );
    sprintf(str, "Right Button\nInterval is %u milliseconds",
           interval);
    MessageBox(hwnd, str, "Double Click", MB_OK);
```

```
        break;
      case WM_DESTROY: // 프로그람을 끝낸다.
        SetDoubleClickTime(OrgDblClkTime); // 시간간격을 본래대로 복귀한다.
        PostQuitMessage(0);
        break;
      default:
        /* 이 switch 문에서 지정된것 이외의 통보문들은
           Windows 2000 이 처리하게 한다. */
        return DefWindowProc(hwnd, message, wParam, lParam);
    }

    return 0;
}
```

이 프로그람의 실행결과는 그림 3-5 와 같다.

그림 3-5. 두번 찰칵프로그람의 실행결과

기타 통보문들

처음에 설명하였던것처럼 Windows 2000 은 이 장에서 설명한것외에도 많은 통보문을 생성한다. 다른 통보문들에 대해서는 이 책의 나머지 부분에서 필요에 따라 설명한다.

제 4 장

차림표의 기초

이 장에서부터 Windows 2000 의 *사용자대면부*에 대한 설명을 시작한다. 일반적으로 Windows 응용프로그람에서는 몇가지 미리 정의된 구성요소들을 리용하여 사용자와 정보를 주고 받는다. 이러한 구성요소들은 Windows 에 정의되여 있으며 그 종류도 다양하다.

이 장에서는 가장 기본적인 구성요소인 차림표에 대해 소개한다. 실제로 작성하는 프로그람에서는 차림표를 사용하는것이 보편적이다. 뒤에서 보게 되지만 차림표의 기본적인 설계는 미리 정의되여 있다. 응용프로그람과 관련되는 고유한 정보만을 추가하면 된다. 차림표가 중요한 구성요소이므로 Windows 2000 은 차림표를 지원한다. 차림표의 기능은 아주 풍부하며 이 장에서는 기초지식만을 설명하고 차림표의 고급한 기능에 대하여서는 제 19 장에서 설명한다.

이 장에서는 자원에 대해서도 소개한다. *자원*이란 프로그람의 외부에 정의되여 있으면서 프로그람에서 사용되는 객체이다. 일반적으로 아이콘, 유표, 비트매프 및 차림표 등이 자원으로 된다. 자원은 거의 모든 Windows 응용프로그람에서 사용되는 중요한 요소이다.

차림표의 기초지식

일반적인 Windows 프로그람의 조작에서 공통적인 요소는 차림표이다. Windows 2000 은 일반적인 세가지 형식의 *차림표*를 지원한다.

- 차림표띠(또는 기본차림표)
- 튀여나오기부분차림표
- 독립된 유동튀여나오기차림표

Windows 응용프로그람에서는 창문의 제일 웃부분에 차림표띠가 표시된다. 이것을 *기본차림표*라고도 한다. *차림표띠*는 응용프로그람의 제일 웃준위의 차림표이다. 차림표띠로부터 부분차림표가 내리펼치기되거나 튀여나오기차림표로서 표시된다. *유동튀여나오기차림표*란 마우스의 오른쪽 단추를 누를 때 표시되는 독립적인 차림표이다.

이 장에서는 차림표띠와 *튀여나오기부분차림표*에 대하여 설명하며 유동튀여나오기차림표에 대해서는 다음 장에서 취급한다. 프로그람의 기본창문에 차림표띠를 추가하는것은 아주 간단하며 아래와 같은 세 단계로 진행한다.

- 자원파일에서 차림표의 구조를 정의한다.
- 프로그람이 기본창문을 작성할 때 차림표를 적재한다.
- 차림표를 선택하였을 때의 처리를 진행한다.

이 장의 나머지 부분들에서는 이 단계들이 실제로 어떻게 진행되는가를 알수 있다.

자 원

Windows 는 몇가지 공통된 형식의 객체를 자원으로 정의한다. 이미 설명한바와 같이 *자원*이란 프로그람에서 리용되는 기본적인 객체이며 프로그람의 외부에서 정의된다. 차림표는 자원의 일종이다. 자원은 프로그람과는 별개로 작성되며 프로그람을 련결할 때 EXE 파일에 추가된다.

자원은 .RC 라는 확장자를 가지는 *자원파일*(*RC 파일*)안에서 정의된다. 소규모의 프

로젝트라면 자원파일의 이름을 EXE 파일의 이름과 같게 정의하는것이 일반적이다. 실례로 프로그람의 이름이 PROG.EXE 라면 자원파일의 이름은 PROG.RC 로 한다. 물론 .RC 라는 확장자만 가진다면 자원파일에 임의의 이름을 붙여도 된다.

자원의 종류에는 일반적인 본문편집기로 작성할수 있는 본문형식도 있다. 본문형식의 자원은 자원파일안에서 정의된다. 아이콘 등과 같은 본문형식이 아닌 자원도 있으며 자원편집기를 사용하여 쉽게 작성할수 있다. 그러나 그 존재를 응용프로그람의 자원파일안에 명기하여야 한다. 이 장에서 작성하는 자원파일은 본문형식뿐이다. 왜냐면 차림표는 본문형식의 자원이기때문이다.

자원파일안에서는 C/C++ 의 명령문을 사용할수 없다. 그대신 자원파일 독자의 명령문을 사용한다. 이 장에서는 차림표를 작성하는데 필요한 명령문들만을 설명하며 그밖의 명령문들은 필요한 때 설명한다.

RC 파일의 번역

자원파일은 그대로는 프로그람에 추가할수 없다. 그 리유는 자원파일을 련결가능한 형식으로 변환하여야 하기때문이다. RC 파일을 작성하면 *자원번역프로그람*으로 번역하여 그것을 RES 파일로 변환한다.(일반적인 자원번역프로그람의 이름은 RC.EXE 이지만 다른 경우도 있다.)

자원파일을 번역하는 방법은 사용하는 번역기에 따라 다르다. 그러나 대표적인 통합개발환경들에서는 이 처리를 자동화하고 있다. 실례로 Microsoft Visual C++에서는 자원파일의 번역과 련결이 자동적으로 진행되게 되여 있다. 여하튼 자원번역프로그람의 출력은 RES 파일이며 프로그람에 련결하여야 한다.

간단한 차림표의 작성

차림표는 자원파일안에서 *MENU* 라는 명령문을 리용하여 정의한다. 모든 차림표에 대해서 아래와 같은 공통적인 구문이 쓰인다.

```
MenuName MENU[options]
{
    menu items
}
```

MenuName 은 차림표의 이름이다. (차림표를 식별하기 위한 옹근수값으로도 할수 있지만 이 책의 실례프로그람에서는 차림표를 참조하기 위해서 이름을 사용한다.) MENU 라는 열쇠단어는 *자원번역프로그람*에 차림표의 작성을 요구한다. Windows 2000

프로그람에서 options 에 지정할수 있는 선택항목은 몇개 되지 않는다. 아래에 가능한 선택항목들을 표시하였다.

선택항목	의　　미
CHARACTERISTICS info	info 에 DWORD 로서 응용프로그람자체의 정보를 지정한다.
LANGUAGE lang, sub-lang	lang 과 sub-lang 에 자원에서 사용하는 언어를 지정한다.
VERSION ver	ver 에 DWORD 로 응용프로그람의 판번호를 지정한다.

간단한 응용프로그람에서는 이 선택항목들을 사용하지 않고 보통 체계설정을 사용한다.

이식과 관련한 요점 : 16bit 의 Windows 3.1 에서는 MENU 명령문에 지정할수 있는 선택항목으로서 PRELOAD 등이 있었다. 이 항목은 Windows 2000 에서는 의미를 가지지 못한다. 그러므로 프로그람을 이식하는 경우에는 이러한 낡은 선택항목을 제거하여야 한다.

차림표를 정의하기 위해서 두개의 명령문 *MENUITEM* 과 *POPUP* 를 리용한다. MENUITEM 은 최종적으로 선택되는 차림표의 항목을 지정한다. POPUP 는 튀여나오기 부분차림표를 지정하며 또한 그 안에 MENUITEM 이나 POPUP 를 포함할수도 있다. 이 명령문들을 사용한 일반적인 구문은 아래와 같다.

MENUITEM “ItemName”, MenID [, Options]
POPUP “PopupName” [, Options]

ItemName 은 [Help] 나 [File] 등과 같은 차림표의 항목을 선택할 때의 이름이다. MenID 는 차림표의 항목과 관련된 유일한 옹근수값으로서 차림표가 선택될 때 응용프로그람에 발송된다.

일반적으로 이 값은 머리부파일안에 마크로로서 정의하여 응용프로그람의 원천파일과 자원파일의 량쪽에 다 포함시킨다. PopupName 은 튀여나오기차림표의 이름이다. 어

느 구문에서도 options 의 값은 표 4-1 에 표시한것과 같다.(이것들은 WINDOWS.H 를 포함시켜 정의한다.)

표 4-1. MENUITEM 과 POPUP 의 선택

선택항목	의 미
CHECKED	차림표항목과 나란히 검사표식이 표시된다.(제일 웃준위차림표에서는 지정할수 없다.)
GRAY	차림표항목이 회색으로 표시되여 선택 불가능한 상태로 된다.
HELP	도움말과 관련된다는것을 표시한다.
INACTIVE	차림표항목선택을 불가능하게 한다.
MENUBARBREAK	차림표띠의 경우는 차림표항목을 개행하여 표시한다. 튀여나오기차림표의 경우는 차림표의 항목을 다른 렬에 표시한다. 이 경우에는 차림표의 항목이 분할띠로 나뉘여진다.
MENUBREAK	분할띠가 사용되지 않는다는 점을 제외하면 MENUBARBREAK 와 같다.
SEPERATOR	분리기로 되는 빈 차림표항목을 작성한다. MENUITEM 에만 지정할수 있다.

뒤에 소개하는 실례프로그람에서 사용되고 있는 차림표의 정의를 아래에서 보여 주었다. 이것은 MENU.RC 라는 파일이다.

```
// 차림표 실례자원파일
#include “menu.h”
MyMenu MENU
{
POPUP “&File”
{
MENUITEM “&Open”, IDM_OPEN
            MENUITEM “&Close”,IDM_CLOSE
            MENUITEM “E&xit”,IDM_EXIT
    }
    POPUP “&Options”
    {
            MENUITEM “&Colors”, IDM_COLORS
            POPUP “&Priority”
```

```
            {
                    MENUITEM "&Low", IDM_LOW
                    MENUITEM "&High", IDM_HIGH
            }
            MENUITEM "&Fonts", IDM_FONT
            MENUITEM "&Resolution", IDM_RESOLUTION
    }
    MENUITEM "&Help", IDM_HELP
}
```

이 차림표의 이름은 MyMenu 이며 [File], [Options], [Help] 세개의 웃준위차림표를 가지고 있다. [File], [Options]에는 튀여나오기부분차림표가 있다. [Priority]라는 항목은 자기의 부분차림표를 가진다.

부분차림표를 가지는 항목에는 차림표 ID 가 없다는데 주의해야 한다. 실제로 선택되는 차림표항목만이 ID 를 가진다. 이 차림표에서는 모든 차림표 ID 값이 IDM 으로 시작되는 마크로로 정의되여 있다. (이 마크로는 MENU.H 라는 머리부파일에 정의되여 있다.) 이런 마크로명은 프로그람과 자원의 중개자로 된다.

차림표항목이름가운데에 있는 &기호는 지름건을 지정하는것이다. 지름건을 사용하면 차림표가 능동상태로 된 때 건을 눌러 차림표항목을 선택할수 있다. 반드시 차림표항목이름앞에 &기호를 사용해야 하는것은 아니지만 다른 문자와 중복되지 않으면 선두에 놓는다. 그러나 [Exit]와 같이 선두문자가 아닌 문자에 사용되는 경우도 있다.

MENU.RC 에서 보여 주는바와 같이 자원파일에는 C/C++형식의 설명문을 포함시킬수도 있다.

MENU.RC 에 포함되여 있는 머리부파일 MENU.H 에는 차림표의 ID 값을 정의하는 마크로가 서술되여 있다. 그것을 아래에 주었다.

```
#define IDM_OPEN                100
#define IDM_CLOSE               101
#define IDM_EXIT                102
#define IDM_COLORS              103
#define IDM_LOW                 104
#define IDM_HIGH                105
#define IDM_FONT                106
#define IDM_RESOLUTION          107
#define IDM_HELP                108
```

이 파일은 매 차림표항목이 선택된 때 돌려 주는 차림표 ID 값을 정의한다. 이 파일은 차림표를 사용하는 프로그람에도 포함된다. 차림표항목에 주어 진 이름이나 값은 중개자로 되며 매개 값은 유일해야 한다. ID 값범위는 1～65535 로 되여야 한다.

프로그람에 차림표를 설치하기

완성된 차림표를 프로그람에 설치하는 가장 간단한 방법은 창문클라스를 작성할 때 차림표이름을 지정하는것이다. 이 경우에는 차림표이름의 문자렬지시자를 보관하는 *lpszMenuName* 을 사용한다. 례하면 MyMenu 라는 이름의 차림표를 사용하면 창문클라스정의를 아래와 같이 한다.

```
wcl.lpszMenuName = “MyMenu”;   // 기본차림표
```

이렇게 하면 MyMenu 는 이 클라스를 기초로 하여 작성되는 모든 창문의 체계설정차림표로 된다. 이것은 이 클라스의 모든 창문이 MyMenu 로 정의된 차림표를 가진다는것을 의미한다.(클라스에서 정의된 차림표를 다른 차림표에 덧쓰기변경하는것도 가능하다.)

차림표선택에 대한 응답

사용자가 차림표를 선택하면 프로그람창문함수에 *WM_COMMAND* 통보문이 발송된다. 이 통보문을 받는 때는 LOWORD(wParam)에 차림표항목의 ID 값이 넣어 져 있다. 즉 LOWORD(wParam)는 RC 파일에 정의된 차림표항목에 대응한 값을 가리키게 된다.

어떤 차림표항목을 선택한 때도 WM_COMMAND 통보문이 발송되며 LOWORD(wParam)에는 차림표항목에 대응한 값이 보관되여 있으므로 선택된 차림표항목을 알려면 겹쌓임 switch 문을 사용하면 된다. 례하면 다음의 프로그람코드는 MyMenu 라는 차림표선택에 응답한다.

```
switch(message) {
  case WM_COMMAND:
    switch(LOWORD(wParam)) {
      case IDM_OPEN:
```

```
            MessageBox(hwnd, "Open File", "Open", MB_OK);
            break;
          case IDM_CLOSE:
            MessageBox(hwnd, "Close File", "Close", MB_OK);
            break;
          case IDM_EXIT:
            response = MessageBox(hwnd, "Quit the Program?",
                                  "Exit", MB_YESNO);
            if(response == IDYES) PostQuitMessage(0);
            break;
          case IDM_COLORS:
            MessageBox(hwnd, "Set Colors", "Colors", MB_OK);
            break;
          case IDM_LOW:
            MessageBox(hwnd, "Low", "Priority", MB_OK);
            break;
          case IDM_HIGH:
            MessageBox(hwnd, "High", "Priority", MB_OK);
            break;
          case IDM_RESOLUTION:
             MessageBox(hwnd, "Resolution Options",
                               "Resolution", MB_OK);
            break;
          case IDM_FONT:
            MessageBox(hwnd, "Font Options","Fonts", MB_OK);
            break;
          case IDM_HELP:
            MessageBox(hwnd, "No Help", "Help", MB_OK);
            break;
        }
    break;
```

설명을 간단히 하기 위하여 매개 차림표항목이 선택된 때의 응답은 단순히 화면에 통보칸을 표시하는것뿐이다. 물론 실제의 응용프로그람에서는 매개 항목에 대하여 고유한 처리를 진행해 주어야 한다.

간단한 차림표프로그람

앞에서 정의한 차림표를 리용한 실례프로그람을 실례 4-1 에 주었다. 이 프로그람의 실행결과는 그림 4-1 에 주었다.

실례 4-1. Menu 프로그람

```
// 차림표의 리용실례

#include <windows.h>
#include <cstring>
#include <cstdio>
#include "menu.h"

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);

char szWinName[] = "MyWin"; // 창문클라스의 이름

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;

  // 창문클라스를 정의한다.
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst;      // 실체의 손잡이
  wcl.lpszClassName = szWinName; // 창문클라스의 이름
  wcl.lpfnWndProc = WindowFunc;  // 창문함수
  wcl.style = 0;                 // 체계설정의 형식

  wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
  wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
  wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

  // 차림표자원을 지정한다.
```

```
wcl.lpszMenuName = "MyMenu";

wcl.cbClsExtra = 0; // 보조기억기령역은 필요 없다.
wcl.cbWndExtra = 0;

// 창문의 배경색은 흰색으로 한다.
wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

// 창문클라스를 등록한다.
if(!RegisterClassEx(&wcl)) return 0;

// 창문을 작성한다.
hwnd = CreateWindow(
  szWinName, // 창문클라스의 이름
  "Introducing Menus", // 창문의 제목
  WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
  CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 창문의 너비는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 창문의 높이는 Windows가 결정하게 한다.
  NULL,          // 어미창문은 없다.
  NULL,          // 차림표는 없다.
  hThisInst,     // 실체의 손잡이
  NULL           // 추가파라메터는 없다.
);

// 창문을 표시한다.
ShowWindow(hwnd, nWinMode);
UpdateWindow(hwnd);

// 통보문순환고리를 작성한다.
while(GetMessage(&msg, NULL, 0, 0))
{
  TranslateMessage(&msg); // 건반통보를 변환한다.
  DispatchMessage(&msg);  // Windows 2000에 조종을 넘긴다.
}

return msg.wParam;
```

```
}

/* 이 함수는 Windows 2000 으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받는다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  int response;

  switch(message) {
    case WM_COMMAND:
      switch(LOWORD(wParam)) {
        case IDM_OPEN:
          MessageBox(hwnd, "Open File", "Open", MB_OK);
          break;
        case IDM_CLOSE:
          MessageBox(hwnd, "Close File", "Close", MB_OK);
          break;
        case IDM_EXIT:
          response = MessageBox(hwnd, "Quit the Program?",
                                "Exit", MB_YESNO);
          if(response == IDYES) PostQuitMessage(0);
          break;
        case IDM_COLORS:
          MessageBox(hwnd, "Set Colors", "Colors", MB_OK);
          break;
        case IDM_LOW:
          MessageBox(hwnd, "Low", "Priority", MB_OK);
          break;
        case IDM_HIGH:
          MessageBox(hwnd, "High", "Priority", MB_OK);
          break;
        case IDM_RESOLUTION:
          MessageBox(hwnd, "Resolution Options",
```

```
                    "Resolution", MB_OK);
            break;
          case IDM_FONT:
            MessageBox(hwnd, "Font Options","Fonts", MB_OK);
            break;
          case IDM_HELP:
            MessageBox(hwnd, "No Help", "Help", MB_OK);
            break;
        }
        break;
      case WM_DESTROY: // 프로그람을 완료한다.
        PostQuitMessage(0);
        break;
      default:
        /* 이 switch 문에서 지정된것 이외의 통보문들은
           Windows 2000 이 처리하게 한다. */
        return DefWindowProc(hwnd, message, wParam, lParam);
    }

    return 0;
  }
```

그림 4-1 은 이 프로그람의 실행결과이다.

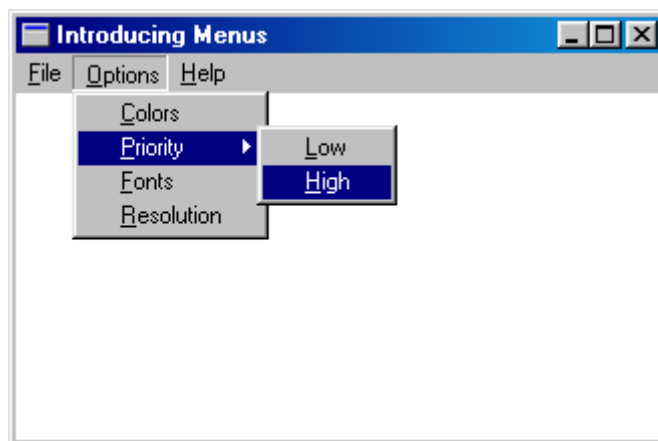



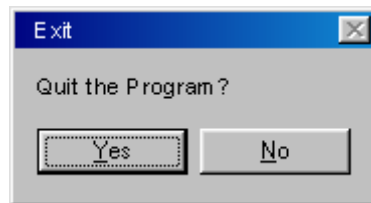
그림 4-1. 차림표프로그람의 실행결과

다시 한보 전진

MessageBox( )의 돌림값의 참조

차림표실례프로그람에서는 사용자가 [Exit]를 선택한 때 다음의 프로그람코드를 실행한다.

```
case IDM_EXIT:
     response = MessageBox(hwnd, "Quit the Program?",
                           "Exit", MB_YESNO);
     if (response == IDYES) PostQuitMessage(0);
     break;
```

이 코드에 의해 아래의 통보칸이 표시된다.

우의 그림을 보면 통보칸에는 [Yes], [No]라는 두개의 단추가 있다. 3장에서 설명한바와 같이 MessageBox( )는 사용자의 응답을 돌려 준다. 이 경우에는 MessageBox( )의 모든 돌림값이 IDYES 또는 IDNO 가 될수 있다. 사용자의 응답이 IDYES 인 경우 프로그람을 완료한다. 그렇지 않은 경우에는 프로그람실행을 계속한다.

이것은 통보칸을 사용하여 사용자에게 두가지 선택안을 주는 경우의 실례이다. Windows 2000 프로그람을 작성할 때 그리 많지 않은 선택안을 사용자에게 주는 상황에서는 통보칸을 사용하는것이 편리하다.

차림표에 지름건을 추가

차림표와 관련하여 자주 사용되는 기능이 Windows 에 있다. 그것은 지름건이다. 지름건이란 차림표항목이 표시되여 있지 않는 때도 차림표항목의 선택을 가능하게 하는

특수한 건입력을 정의한것이다.

다시말하여 차림표조작을 하지 않고도 지름건을 눌러 차림표항목을 선택할수 있다는 것이다. 지름건이라는 용어는 아주 적절한 표현이다. 왜냐하면 차림표를 능동상태로 하고 항목을 선택하기 보다 건을 누르는것이 조작이 빠르기때문이다.

차림표에 대응한 지름건을 정의하기 위해서는 자원파일에 지름건표를 추가한다. 이 지름건표는 일반적으로 다음과 같은 구문형식으로 서술된다.

```
TableName  ACCELERATORS  [ accel-options]
{
    Key1,MenuID1 [ , type ] [ options ]
    Key2,MenuID2 [ , type ] [ options ]
    Key3,MenuID3 [ , type ] [ options ]
    ...
    ...
    KeyN,MenuIDN [ , type ] [ options ]
}
```

TableName 은 지름건표의 이름이다. *ACCELERATORS 명령문*에는 MENU 명령문과 같은 추가선택을 지정할수 있다. 필요하다면 그것을 accel-options 에 지정한다. 그러나 대부분의 응용프로그람에서는 체계설정을 리용한다.

지름건표내부에서 Key 에는 차림표항목을 선택하는 건을 지정하고 MenuID 에는 대응하는 차림표항목의 ID 를 지정한다. type 에는 건이 표준건(이 설정은 체계설정이다.)인가, 가상건인가를 지정한다.

options 에는 *NOINVERT*, *ALT*, *SHIFT*, CONTROL 중의 어느 하나를 지정한다. NOINVERT 를 지정하면 지름건을 눌러도 차림표항목이 강조표시되지 않게 된다. *ALT* 는 [Alt]건, SHIFT 는 [Shift]건, *CONTROL* 은 [Ctrl]건을 지정하는것이다.

Key값으로는 인용괄호(“)안에 들어 있는 문자, 건의 ASCII코드를 가리키는 옹근수값 또는 가상건코드를 가리키는 옹근수값을 지정할수 있다. 인용괄호안에 들어 있는 문자가 지정된 경우에는 이것이 ASCII 문자인가를 판단한다. 옹근수값인 경우에는 type 에 ASCII 를 지정하여 자원번역프로그람에 ASCII 코드를 사용한다는것을 명백하게 지정해주어야 한다. 가상건인 경우에는 type 에 *VIRTKEY* 를 지정해 주어야 한다.

인용괄호안에 들어 있는 문자가 대문자인 경우에는 [Shift]건과 동시에 이 건을 누르면 차림표항목이 선택된다. 소문자인 경우에는 이 건만을 누르는것으로서 차림표항목이 선택된다. 건이 소문자로 지정되고 추가선택에 ALT 가 지정된 경우에는 [Alt]건과 그 건을 동시에 누르는것으로서 차림표항목이 선택된다.(건이 대문자이고 ALT 가 지정된 경우에는 차림표항목을 선택하자면 [Shift]건과 [Alt]건을 동시에 눌러야 한다.) 사용자에게 [Ctrl]건과 문자를 동시에 누르게 하려는 경우에는 건의 앞에 ^를 지정한다.

제 3 장에서 설명한바와 같이 *가상건*은 체계에 존재하지 않는 건코드이다. 지름건으로서 가상건을 사용하려면 이것을 가리키는 마크로를 Key 에 지정하고 VIRTKEY 를 type 에 지정한다. 임의의 건조합을 실현하기 위해 ALT, SHIFT, CONTROL 을 지정할 수도 있다. 몇가지 실례를 아래에 준다.

```
"A", IDM_X                  // [Shift]+[A]을 누르면 선택된다.
"a", IDM_X                  // [a]를 누르면 선택된다.
"^a", IDM_X                 // [Ctrl]+[a]를 누르면 선택된다.
"a", IDM_X, ALT             // [Alt]+[a]를 누르면 선택된다.
VK_F2, IDM_X                // [F2]를 누르면 선택된다.
VK_F2, IDM_X, SHIFT         // [Shift]+[F2]를 누르면 선택된다.
```

MyMenu 에 지름건을 추가한 자원파일 MENU.RC 의 내용은 다음과 같다.

```
//차림표와 지름건의 자원파일 실례
#include "menu.h"
MyMenu MENU
{
POPUP "&File"
{
MENUITEM "&Open\tF2", IDM_OPEN
            MENUITEM "&Close\tF2",IDM_CLOSE
            MENUITEM "E&xit\tCtrl+X",IDM_EXIT
    }
    POPUP "&Options"
    {
            MENUITEM "&Colors\tCtrl+C", IDM_COLORS
            POPUP "&Priority"
            {
                    MENUITEM "&Low\tF4", IDM_LOW
                    MENUITEM "&High\tF5", IDM_HIGH
            }
            MENUITEM "&Fonts\tCtrl+F", IDM_FONT
            MENUITEM "&Resolution\tCtrl+R", IDM_RESOLUTION
    }
    MENUITEM "&Help", IDM_HELP
}
// 차림표의 지름건을 정의한다.
```

```
{
    VK_F2, IDM_OPEN, VIRTKEY
    VK_F3, IDM_CLOSE, VITRKEY
    "^X", IDM_EXIT
    "^C", IDM_COLORS
    VK_F4, IDM_LOW, VIRTKET
    VK_F5, IDM_HIGH, VIRTKEY
    "^F", IDM_FONT
    "^R", IDM_RESOLUTION
    VK_F1, IDM_HELP, VIRTKEY
}
```

차림표정의안에 지름건이 추가되였다는데 주목해야 한다. 매개 차림표항목에서는 타브를 사용하여 지름건의 표시위치를 분리한다. WINDOWS.H머리부파일을 포함시키는것은 그 안에 가상건의 마크로가 정의되여 있기때문이다.

지름건표의 적재

자원파일에 차림표와 함께 지름건을 서술하였다면 프로그람에서 *LoadAccelerators( )*라는 API 함수를 사용하여 지름건을 적재하여야 한다. 아래에 이 함수의 선언을 보여 주었다.

```
HACCEL LoadAccelerators(HINSTANCE ThisInst, LPCSTR Name);
```

ThisInst는 응용프로그람실체의 손잡이이며 Name은 지름건표의 이름이다. 이 함수는 지름건표의 손잡이를 돌려 주며 표가 적재되지 않는 경우에는 NULL을 돌려 준다.

창문이 작성된 다음에 LoadAccelerators( )를 호출해야 한다. 실례로 MyMenu라는 지름건표를 넣는 방법을 아래에 주었다.

```
HACCEL hAccel;
hAccel = LoadAccelerators(hThisInst, "MyMenu");
```

hAccel값은 후에 지름건을 처리할 때 리용된다.

지름건의 변환

LoadAccelerators( )로 지름건을 적재하여도 통보문순환고리에 다른 하나의 API 함수를 추가하지 않고서는 지름건을 처리할수 없다. 이 함수를 *TranslateAccelerator( )*라고 한다. 아래에 선언을 주었다.

int TranslateAccelerator(HWND hwnd, HACCEL hAccel, LPMSG lpMess);

hwnd 는 지름건을 변환하는 대상으로 되는 창문의 손잡이이다. hAccel 은 사용하는 지름건표의 손잡이이다. 이 손잡이는 LoadAccelerators( )에서 돌려 받는것이다. 마감의 lpMess 는 통보문에 대한 지시자이다. TranslateAccelerator( )는 지름건이 눌리웠을 때 TRUE 를 돌려 주며 그렇지 않으면 FALSE 를 돌려 준다.

TranslateAccelerator( )는 지름건의 입력을 이에 대응하는 WM_COMMAND 통보문으로 변환하여 통보문을 창문에 보낸다. 이 통보문에서는 LOWORD(wParam)에 지름건에 대응하는 ID 값이 보관되여 있다. 따라서 프로그람에서는 WM_COMMAND 통보문이 차림표선택에 의하여 생성되는것처럼 볼수 있다.

TranslateAccelerator( )는 지름건이 눌리울 때마다 WM_COMMAND 통보문을 발송하므로 이때 프로그람에서는 *TranslateMessage( )*나 DispatchMessage( )를 실행할 필요는 없다. 그러므로 TranslateAccelerator( )를 사용하는 경우의 통보문순환고리는 아래와 같다.

```
while(GetMeassage(&msg, NULL, 0, 0 ))
{
    if (! TranslateAccelerator(hwnd, hAccel, &msg)) {
            TranslateMessage(&msg);    // 건반통보를 변환한다.
            DispatchMessage( &msg);    // Windows 2000 에 조종을 넘긴다.
    }
}
```

지름건의 시험

지름건의 사용법을 시험해보기 위하여 실례 4-2 의 WinMain( )을 앞의 프로그람의 것과 교체하고 자원파일에 지름건표를 추가한다.

실례 4-2. Accel 프로그람

```
// 지름건의 처리

#include <windows.h>
#include <cstring>
#include <cstdio>
#include "menu.h"
```

```
LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);

char szWinName[] = "MyWin"; // 창문클라스의 이름

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;
  HACCEL hAccel;

  // 창문클라스를 정의한다.
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst;      // 실체의 손잡이
  wcl.lpszClassName = szWinName; // 창문클라스의 이름
  wcl.lpfnWndProc = WindowFunc;  // 창문함수
  wcl.style = 0;                 // 체계설정형식

  wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
  wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
  wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

  // 차림표자원을 지정한다.
  wcl.lpszMenuName = "MyMenu";

  wcl.cbClsExtra = 0; // 보조기억기령역은 필요 없음
  wcl.cbWndExtra = 0;

  // 창문의 배경색을 흰색으로 한다.
  wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

  // 창문클라스를 등록한다.
  if(!RegisterClassEx(&wcl)) return 0;

  // 창문을 작성한다.
  hwnd = CreateWindow(
```

```
    szWinName, // 창문클라스의 이름
    "Adding Accelerator Keys", // 제목
    WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준형식으로 한다.
    CW_USEDEFAULT, // X 자리표는 Windows가 결정하게 한다.
    CW_USEDEFAULT, // Y 자리표는 Windows가 결정하게 한다.
    CW_USEDEFAULT, // 창문의 너비는 Windows가 결정하게 한다.
    CW_USEDEFAULT, // 창문의 높이는 Windows가 결정하게 한다.
    NULL,          // 어미창문은 없다.
    NULL,          // 차림표는 없다.
    hThisInst,     // 실체의 손잡이
    NULL           // 추가파라메터는 없다.
  );

  // 건반가속기를 적재한다.
  hAccel = LoadAccelerators(hThisInst, "MyMenu");

  // 창문을 표시한다.
  ShowWindow(hwnd, nWinMode);
  UpdateWindow(hwnd);

  // 통보문순환고리를 작성한다.
  while(GetMessage(&msg, NULL, 0, 0))
  {
    if(!TranslateAccelerator(hwnd, hAccel, &msg)) {
      TranslateMessage(&msg); // 건반통보를 변환한다.
      DispatchMessage(&msg);  // Windows 2000에 조종을 넘긴다.
    }
  }

  return msg.wParam;
}
```

다시 한보 전진

WM_COMMAND의 상세

지금까지 설명한바와 같이 차림표항목을 선택하거나 지름건을 누르면 *WM_COMMAND* 통보문이 발송되고 그때 LOWORD(wParam)에는 선택된 차림표항목 또는 지름건의 ID 가 보관된다. 그러나 LOWORD(wParam)값만으로는 어떤 사건이 발생하였는가를 구별할수 없다. 일반적인 경우에는 사용자가 차림표항목을 선택하는가 지름건을 누르는가 하는것이 문제로 되지 않는다. 하지만 Windows는 wParam의 웃단어에 이 사건을 식별하기 위한 정보를 제공한다. HIWORD(wParam)의 값이 0 이면 사용자가 차림표항목을 선택한것으로 된다.

그 값이 1이면 사용자가 지름건을 누른것으로 된다.

시험을 목적으로 아래의 프로그람코드를 차림표프로그람과 치환하시오. [Open]이 차림표 또는 지름건가운데 어느것으로 선택되였는가가 통보칸에 표시된다.

```
case IDM_OPEN:
    if (HIWORD(wParam))
        MessageBox(hwnd,"Open File via Accelerator", "Open", MB_OK");
    else
        MessageBox(hwnd,"Open File via Menu Selection", "Open",
                    MB_OK");
    break;
```

WM_COMMAND 통보문의 lParam 의 값은 차림표항목이나 지름건의 선택에는 관계없이 NULL로 되므로 사용되지 않는다.

다음 장에서 설명하지만 WM_COMMAND 통보문은 사용자가 여러가지 조종체들을 조작할 때도 생성된다. 이 경우에는 wParam, lParam 의 의미가 여기에서 설명한것과 약간 다르다. 례하면 lParam 값은 조종체의 손잡이로 된다.

차림표에 대응되지 않는 지름건

건가속은 주로 차림표항목을 재빠르게 선택하기 위한 수단으로 사용되지만 그것만으로 그의 역할이 제한되는것은 아니다. 례를 든다면 차림표항목에 대응되지 않는 지름건

을 정의할수도 있다. 이러한 건은 건반마크로를 사용하는 경우나 빈번히 사용되는 추가 선택기능을 선택하는것으로 된다. 차림표에 대응되지 않는 지름건을 정의하려면 간단히 거기에 유일한 값을 붙여 지름건표에 추가하기만 하면 된다.

실례로 차림표에 대응되지 않는 지름건을 차림표프로그람에 추가해 보자. 이 건을 [Ctrl]+[T]로 하여 그것이 눌리울 때마다 현재시간과 날자를 통보칸에 표시하도록 해 보자. C/C++표준함수로 현재시간과 날자를 얻는다. 우선 다음과 같이 건표를 변경한다.

실례 4-3. NonMenuAccel 프로그람

```
MyMenu ACCELERATORS
{
            VK_F2, IDM_OPEN, VIRTKEY
    VK_F3, IDM_CLOSE, VITRKEY
            "^X", IDM_EXIT
    "^C", IDM_COLORS
    VK_F4, IDM_LOW, VIRTKET
            VK_F5, IDM_HIGH, VIRTKEY
    "^R", IDM_RESOLUTION
            "^F", IDM_FONT
    VK_F1, IDM_HELP, VIRTKEY
            "^T", IDM_TIME
}
```

다음에 아래 한행을 MENU.H에 추가한다.

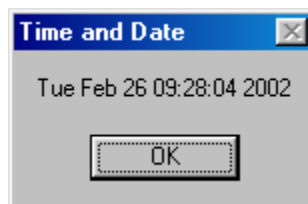
```
#define IDM_TIME 500
```

마지막으로 아래의 WindowFunc( )를 차림표프로그람에 있는것과 바꾸어놓는다. 또한 프로그람코드의 첫머리에 <ctime.h>라는 머리부파일을 포함시킨다.

```
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  int response;
  struct tm *tod;
  time_t t;
  char str[80];
```

```
switch(message) {
  case WM_COMMAND:
    switch(LOWORD(wParam)) {
      case IDM_OPEN:
        MessageBox(hwnd, "Open File", "Open", MB_OK);
        break;
      case IDM_CLOSE:
        MessageBox(hwnd, "Close File", "Close", MB_OK);
        break;
      case IDM_EXIT:
        response = MessageBox(hwnd, "Quit the Program?",
                              "Exit", MB_YESNO);
        if(response == IDYES) PostQuitMessage(0);
        break;
      case IDM_COLORS:
        MessageBox(hwnd, "Set Colors", "Colors", MB_OK);
        break;
      case IDM_LOW:
        MessageBox(hwnd, "Low", "Priority", MB_OK);
        break;
      case IDM_HIGH:
        MessageBox(hwnd, "High", "Priority", MB_OK);
        break;
      case IDM_RESOLUTION:
         MessageBox(hwnd, "Resolution Options",
                          "Resolution", MB_OK);
        break;
      case IDM_FONT:
        MessageBox(hwnd, "Font Options","Fonts", MB_OK);
        break;
      case IDM_TIME: // 시간을 표시한다.
        t = time(NULL);
        tod = localtime(&t);
        strcpy(str, asctime(tod));
        str[strlen(str)-1] = '\0'; // \r\n을 제거한다.
        MessageBox(hwnd, str, "Time and Date", MB_OK);
        break;
```

```
          case IDM_HELP:
            MessageBox(hwnd, "No Help", "Help", MB_OK);
            break;
        }
        break;
      case WM_DESTROY: // 프로그람을 완료한다.
        PostQuitMessage(0);
        break;
      default:
        /* 이 switch 문에서 지정된것 이외의 통보문은
           Windows 2000 이 처리하게 한다. */
        return DefWindowProc(hwnd, message, wParam, lParam);
    }

    return 0;
}
```

이 프로그람을 실행하여 [Ctrl]+[T]를 누르면 다음과 같은 통보칸이 표시된다.

클라스차림표의 덧쓰기

지금까지 작성한 프로그람에서는 기본차림표가 WNDCLASSEX 구조체의 lpszMenuName 성원으로 지정되여 있었다. 이것은 이 클라스에 기초하여 작성되는 모든 창문에서 사용되는 *클라스차림표*를 지정하는것이며 간단한 프로그람의 기본차림표를 지정하는데 쓰이는 방법이다. 그러나 *CreateWindow( )*를 사용하여 기본차림표를 지정하는 방법도 쓰인다. 제 2 장에서 본것처럼 CreateWindow( )는 다음과 같은 파라메터를 가진다.

```
HWND CreateWindow(
            LPCSTR lpClassName,            // 창문클라스이름
```

```
            LPCSTR lpWinName,           // 창문제목
            DWORD dwStyle,              // 창문형식
            int X, int Y,               // 창문의 왼쪽 웃좌표
            int Width, int Height,      // 창문의 크기
            HWND hParent,               // 어미창문의 손잡이
            HMENU hMenu,                // 기본차림표의 손잡이
            HINSTANCE hThisInst         // 실체의 손잡이
      LPVOID lpszAdditional             // 보충정보지시자
   );
```

hMenu 라는 파라메터에 주목해야 한다. 이 파라메터는 창문작성시에 기본차림표를 지정하는데 사용된다. 지금까지의 프로그람들에서는 이 파라메터에 NULL 이 지정되여 있었다. hMenu 가 NULL 인 경우에는 클라스의 차림표가 사용된다. 그러나 이 파라메터에 차림표의 손잡이를 지정하면 그 차림표가 창문의 기본차림표로서 리용된다. 이런 경우에는 hMenu 에서 지정된 차림표가 클라스차림표를 덧쓰기하는것으로 된다.

이 책에서 소개하는 간단한 프로그람들에서는 클라스차림표를 덧쓰기할 필요는 없지만 여하튼 덧쓰기가 필요하게 되는 경우도 앞으로 있을수 있다. 례하면 일반적인 창문클라스를 정의해두고 그것을 특정한 용도에서만 변경하는것과 같은 때이다.

CreateWindow( )를 사용하여 기본차림표를 지정하려면 차림표의 손잡이가 필요하다. 이것을 얻기 위한 가장 간단한 방법은 *LoadMenu( )*라는 API 함수를 호출하는것이다. 아래에 선언을 보여 주었다.

```
HMENU LoadMenu(HINSTANCE hInst, LPCSTR lpName);
```

hInst 는 응용프로그람실체의 손잡이이다. 차림표이름에 대한 지시자가 lpName 에 주어 진다. LoadMenu( )는 차림표의 손잡이를 돌려 주며 호출이 실패하면 NULL 을 돌려 준다. 차림표의 손잡이를 얻으면 그것을 CreateWindow( )의 hMenu 파라메터에 설정한다.

LoadMenu( )를 사용하여 차림표를 적재한다는것은 기억기를 확보할 객체를 작성하는것으로 된다. 이 기억기는 프로그람이 완료되기전에 해제되여야 한다. 기억기가 창문에 관련되여 있는 경우 프로그람이 완료될 때 기억기의 해제가 자동적으로 진행된다. 그렇지 않은 경우에는 수동적으로 기억기를 해제시켜주어야 한다. 이를 위해서 *DestroyMenu( )*라는 API 함수를 사용한다. 아래에 선언을 보여 주었다.

```
BOOL DestroyMenu(HMENU hMenu);
```

hMenu 는 파괴할 차림표의 손잡이이다. 이 함수는 호출이 성공하면 0 이 아닌 값을

돌려 주며 실패하면 0 을 돌려 준다. 이미 설명한바와 같이 적재한 차림표가 창문과 관련되여 있다면 DestroyMenu( )를 사용할 필요가 없다.

클라스차림표를 덧쓰기하는 실례프로그람

클라스차림표가 어떻게 덧쓰기되는가를 보기 위하여 지금까지 작성한 프로그람을 개조해 보자. 우선 아래에 보여 주는바와 같이 MENU.RC 파일에 PlaceHolder 라는 이름의 두번째 차림표를 추가한다.

```
// 두번째 차림표의 정의
#include <windows.h>
#include :"menu.h"
// 클라스차림표로 되는 차림표
PlaceHolder MENU
{
     POPUP "&File"
     {
        MENUITEM "&Exit\tCtrl+X", IDM_EXIT
     }
     MENUITEM "&Help", IDM_HELP
}

// CreateWindow( ) 에서 사용되는 차림표
MyMenu MENU
{
     POPUP "&File"
     {
             MENUITEM "&Open\tF2", IDM_OPEN
             MENUITEM "&Close\tF3", IDM_CLOSE
             MENUITEM "E&xit\tCtrl+X", IDM_EXIT
     }
     POPUP "&Options"
     {
                         MENUITEM "&Colors\tCtrl+C", IDM_COLORS
                         POPUP "&Priority"
                         {
                                   MENUITEM "&Low\tF4",IDM_LOW
```

```
                                        MENUITEM "&High\tF5",IDM_HIGH
                                }
                                MENUITEM "&Font\tCtrl+F",IDM_FONT
                                MENUITEM "&Resolution\Ctrl+R", IDM_RESOLUTION
        }
        MENUITEM "&Help", IDM_HELP
}

// 차림표의 지름건을 정의한다.
MyMenu ACCELERATORS
{
        VK_F2, IDM_OPEN , VIRTKEY
                VK_F3, IDM_CLOSE, VIRTKEY
                "^X", IDM_EXIT
                "^C", IDM_COLORS
                VK_F4, IDM_LOW, VIRTKEY
                VK_F5, IDM_HIGH,VIRTKEY
                "^F", IDM_FONT
                "^R", IDM_RESOLUTION
        VK_F1, IDM_HELP, VIRTKEY
        "^T", IDM_TIME
}
```

PlaceHolder 차림표는 클라스차림표로서 사용된다. 다시말하면 이것이 WNDCLASSEX 의 lpszMenuName 성원에 설정된다는것이다. MyMenu 는 독립적으로 적재되여 그의 손잡이가 CreateWindow( )의 hMenu 파라메터에 설정된다. 그리하여 MyMenu 가 PlaceHolder 를 덧쓰기 하게 된다.

MENU.H 의 내용은 아래와 같다. 이것은 IDM_TIME 을 추가한것을 내놓고는 앞절에서 작성한것과 같다.

```
#define IDM_OPEN                100
#define IDM_CLOSE               101
#define IDM_EXIT                102
#define IDM_COLORS              103
#define IDM_LOW                 104
#define IDM_HIGH                105
#define IDM_FONT                106
#define IDM_RESOLUTION          107
```

```
#define IDM_HELP                108
#define IDM_TIME                500
```

이 장에서 차림표프로그람에 많은 변경을 가하였으므로 클라스차림표를 덧쓰는 프로그람전체를 실례 4-4 에 표시한다. 이 프로그람은 이 장에서 설명한 모든 내용을 담고 있다.

실례 4-4. OverRide 프로그람

```
// 클라스차림표의 덧쓰기

#include <windows.h>
#include <cstring>
#include <cstdio>
#include <ctime>
#include "menu.h"

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);

char szWinName[] = "MyWin"; // 창문클라스의 이름

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;
  HACCEL hAccel;
  HMENU hMenu;

  // 창문을 정의한다.
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst;      // 실체의 손잡이
  wcl.lpszClassName = szWinName; // 창문클라스의 이름
  wcl.lpfnWndProc = WindowFunc;  // 창문함수
```

```
wcl.style = 0;                    // 체계설정형식

wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

// 차림표자원을 지정한다. 이것은 덧쓰기된다.
wcl.lpszMenuName = "PlaceHolder";

wcl.cbClsExtra = 0; // 보조기억기령역은 필요 없다.
wcl.cbWndExtra = 0;

// 창문의 배경색은 흰색으로 한다.
wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

// 창문클라스를 등록한다.
if(!RegisterClassEx(&wcl)) return 0;

// 수동으로 기본차림표를 적재한다.
hMenu = LoadMenu(hThisInst, "MyMenu");

// 창문을 작성한다.
hwnd = CreateWindow(
  szWinName, // 창문클라스의 이름
  "Adding Accelerator Keys", // 제목
  WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
  CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 창문의 너비는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 창문의 높이는 Windows가 결정하게 한다.
  NULL,          // 어미창문은 없다.
  hMenu,         // 다른 기본차림표로 치환한다.
  hThisInst,     // 실체의 손잡이
  NULL           // 추가파라메터는 없다.
);

// 건반가속기를 적재한다.
hAccel = LoadAccelerators(hThisInst, "MyMenu");
```

```
    // 창문을 표시한다.
    ShowWindow(hwnd, nWinMode);
    UpdateWindow(hwnd);

    // 통보문순환고리를 작성한다.
    while(GetMessage(&msg, NULL, 0, 0))
    {
      if(!TranslateAccelerator(hwnd, hAccel, &msg)) {
        TranslateMessage(&msg); // 건반통보를 변환한다.
        DispatchMessage(&msg);  // Windows 2000에 조종을 넘긴다.
      }
    }

    return msg.wParam;
}

/* 이 함수는 Windows 2000으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받는다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
    int response;
    struct tm *tod;
    time_t t;
    char str[80];

    switch(message) {
      case WM_COMMAND:
        switch(LOWORD(wParam)) {
          case IDM_OPEN:
            MessageBox(hwnd, "Open File", "Open", MB_OK);
            break;
          case IDM_CLOSE:
            MessageBox(hwnd, "Close File", "Close", MB_OK);
            break;
          case IDM_EXIT:
```

```
        response = MessageBox(hwnd, "Quit the Program?",
                              "Exit", MB_YESNO);
        if(response == IDYES) PostQuitMessage(0);
        break;
      case IDM_COLORS:
        MessageBox(hwnd, "Set Colors", "Colors", MB_OK);
        break;
      case IDM_LOW:
        MessageBox(hwnd, "Low", "Priority", MB_OK);
        break;
      case IDM_HIGH:
        MessageBox(hwnd, "High", "Priority", MB_OK);
        break;
      case IDM_RESOLUTION:
         MessageBox(hwnd, "Resolution Options",
                          "Resolution", MB_OK);
        break;
      case IDM_FONT:
        MessageBox(hwnd, "Font Options","Fonts", MB_OK);
        break;
      case IDM_TIME: // 시간을 표시한다.
        t = time(NULL);
        tod = localtime(&t);
        strcpy(str, asctime(tod));
        str[strlen(str)-1] = '\0'; // \r\n을 제거한다.
        MessageBox(hwnd, str, "Time and Date", MB_OK);
        break;
      case IDM_HELP:
        MessageBox(hwnd, "No Help", "Help", MB_OK);
        break;
    }
    break;
  case WM_DESTROY: // 프로그람을 완료한다.
    PostQuitMessage(0);
    break;
  default:
    /* 이 switch 문에서 지정되지 않은 통보문은
       Windows 2000에 처리를 맡긴다. */
```

```
            return DefWindowProc(hwnd, message, wParam, lParam);
        }

        return 0;
    }
```

WinMain( )함수안의 프로그람코드에 특히 주목해야 한다. 클라스차림표로서 PlaceHolder 를 지정한 창문클라스가 작성되고 있다. 그러나 실제로는 창문이 작성되기 전에 MyMenu 가 적재되여 그것의 손잡이가 CreateWindow( )를 호출하는데 리용되므로 클라스차림표가 덧쓰기되고 MyMenu 가 표시된다.

이 프로그람을 사용하여 다른 실험을 해볼 필요가 있다. 실레로 클라스차림표가 덧쓰기된다면 처음부터 지정할 필요는 없지 않는가 하는것을 알아 보는 실험이다. 이것을 확인해 보기 위해 lpszMenuName의 값을 NULL로 한다. 이렇게 해도 프로그람에는 아무런 영향도 가해 지지 않는다.

이 실례프로그람에서는 MyMenu 와 PlaceHolder 가 다 같은 창문함수에서 처리할 수 있는 차림표로 되여 있다. 그러므로 량자는 동일한 차림표 ID 의 모임을 사용한다.(그러나 PlaceHolder에는 두개의 차림표항목밖에 없다.)

이러한 기능은 앞에서 본 프로그람에서 어떠한 차림표의 사용도 가능하게 한다. 덧쓰기하는 차림표의 형식이나 구조에 한정되지 않는다. 그러나 어떤 차림표가 사용된다고 해도 창문함수에서 적절한 응답을 할수 있게 하는것이 중요하다.

CreateWindow( )보다 WNDCLASSEX 를 사용하는 쪽이 차림표의 작성이 간단하다. 이 책의 나머지 프로그람들에서는 간단한 방법을 리용한다.

또 하나의 중요한 요점이 있다. 그것은 MyMenu 가 CreatWindow( )에 의해 작성된 창문과 관련되여 있으므로 프로그람을 완료할 때 차림표가 자동적으로 파괴되며 DestroyMenu( )를 호출할 필요는 없다는것이다.

제 5 장

대화칸

차림표는 일반적인 Windows 2000 응용프로그람의 매우 중요한 입력수단이지만 모든 형식의 사용자입력을 처리할수 있는 만능의 수단은 아니다. 례하면 차림표로 시간이나 날자를 입력하는것은 불가능하다. 이러한 형식의 입력을 수행하기 위하여 Windows 에서는 대화칸을 리용한다.

*대화칸*이란 사용자와 응용프로그람의 대화를 위한 다양한 입력수단을 제공하는 특수한 창문이다. 일반적으로 대화칸은 차림표로서는 실현이 어렵거나 불가능한 정보의 선택이나 입력을 가능하게 하는데 리용된다. 이 장에서는 대화칸의 작성방법과 조작법을 설명한다.

대화칸은 *조종체*를 가지고 사용자와 대화한다. 어떤 의미에서 대화칸은 여러가지 조종체를 넣어 두는 그릇이라고 말할수 있다. 이 장에서는 Windows 의 표준적인 조종체들가운데서 세가지를 설명한다.

또한 대화칸과 몇가지 조종체의 사용법을 보여 주기 위하여 간단한 자료기지응용프로그람을 작성한다. 이 자료기지는 수십권의 서적의 제목, 저자명, 출판사이름, 저작권, 날자를 보관한다. 이 장에서 작성하는 대화칸에서는 제목을 선택하여 그 서적에 대한 다른 정보를 얻을수 있게 되여 있다. 자료기지의 기능은 최소한의 단순한것이며 실제 응용프로그람에서 대화칸을 효과적으로 사용하는 방법을 보여 주는데 목적을 둘뿐이다.

대화칸은 조종체를 사용한다

대화칸은 그자체로서는 아무런 기능도 수행하지 못한다. 기술적으로 말하면 대화칸은 단순한 관리장치이다. 사용자와 대화하는것은 대화칸내부에 있는 조종체이다. 정확히 말하면 조종체란 입출력을 진행하기 위한 특수한 창문이다. 조종체는 어미창문에 속해있으며 그 어미창문이 이 장에서는 대화칸으로 된다.

Windows 2000 은 다양한 표준적조종체들을 제공하고 있다. 여기에는 *누름단추*, 검사칸, 단일선택단추, *목록칸*, *편집칸*, 복합칸, 흘림띠 및 정적조종체 등이 있다.(다음장에서 설명하지만 Windows 2000 은 *공통조종체*라고 하는 몇가지 고급한 조종체도 제공한다.)

이 장의 실례프로그람에서는 표준조종체중에서 누름단추, 목록칸, 편집칸만을 리용한다.

누름단추는 사용자가 마우스를 찰칵할 때 어떤 응답을 주는 조종체이다. 이미 통보칸에서 누름단추를 사용하였다. 례하면 통보칸에서 사용한 [OK]단추는 누름단추이다.

목록칸은 사용자가 하나 혹은 그이상의 항목을 선택할수 있는 목록을 표시하므로 례하면 파일이름을 표시하는 등의 목적에 자주 사용된다.

편집칸은 사용자의 문자렬입력을 받아 들인다. 편집칸은 여러가지 본문편집기능도 제공한다. 따라서 문자렬을 입력하는 프로그람에서는 간단히 편집칸을 표시해 놓고 사용자가 문자렬입력을 마칠 때까지 대기하도록 하면 좋다.

조종체는 통보문을 생성하며(사용자가 조작할 때) 한편으로는 통보문을 받는다.(응용프로그람으로부터 통보문을 보내온다.) 이것은 중요한 내용이다. 조종체가 생성한 통보문은 사용자가 진행한 조작내용을 가리키게 된다. 조종체에 보내는 통보문은 기본적으로는 조종체에 대한 명령으로 된다.

양식화대화칸과 비양식화대화칸

대화칸에는 양식화형과 비양식화형의 두가지 형태가 있다. 일반적으로 사용되는것은 *양식화대화칸*이다. 양식화대화칸은 프로그람조작을 계속하기 위하여 사용자의 응답을 요구한다.

양식화대화칸이 능동상태로 되여 있는 때는 대화칸을 닫기전에 응용프로그람의 다른 부분에 입력초점을 이동할수 없다. 보다 정확히 말하면 양식화대화칸의 *어미창문*은 대화칸을 닫을 때까지는 능동상태로 되지 못한다.(일반적으로 어미창문이란 대화칸을 연 창문이다.)

*비양식화대화칸*은 프로그람의 다른 부분이 리용되는것을 막지 않는다. 따라서 프로그람의 다른 부분에로 초점을 옮기기 위해 대화칸을 닫을 필요가 없다. 비양식화대화칸의 어미창문은 계속 능동상태에 있다. 즉 비양식화대화칸은 양식화대화칸보다 독립성이 강하다.

일반적으로 자주 사용되는 양식화대화칸에 대하여 먼저 설명하기로 한다. 비양식화대화칸의 실례프로그람도 이 장에서 준다.

대화칸의 통보문처리

대화칸은 창문의 일종이다. 그 내부에서 발생한 사건은 기본창문의 통보문과 같은 방식으로 프로그람에 전송된다. 그러나 대화칸의 통보문은 프로그람에 있는 기본창문의 창문함수에 전송되지 않는다. 대신에 매개 대화칸은 자기의 창문함수를 가지며 이 함수를 일반적으로 *대화함수* 혹은 *대화수속*이라고 한다. 이 함수는 다음과 같이 선언된다.(함수이름은 임의로 정의해도 상관없다.)

```
BOOL CALLBACK DFunc(HWND hdwnd, UINT message,
                    WPARAM wParam, LPARAM lParam);
```

보는바와 같이 대화함수에는 프로그람의 기본창문의 창문함수와 꼭 같은 파라메터가 주어 진다. 그러나 돌림값으로서 TRUE 또는 FALSE 를 돌려 준다는것이 기본창문의 창문함수와 다르다.

프로그람의 기본창문의 창문함수와 마찬가지로 대화칸의 창문함수도 수많은 통보문을 받는다. 통보문을 처리하였다면 TRUE 를 돌려 주어야 하며 통보문을 처리하지 않은 경우에는 FALSE 를 돌려 주어야 한다.

대화칸안에 있는 매개 조종체에는 각각 자체의 자원 ID가 주어 진다. 사용자가 조종체를 조작하면 WM_COMMAND 통보문이 대화함수에 전송되여 사용자가 조작한 조종체의 ID 와 조작내용이 전달된다. 대화함수는 통보문의 내용을 참조하여 적절한 처리를 진행한다. 통보문을 처리하는 방법은 프로그람의 기본창문의 창문함수와 꼭같다.

대화칸을 열기

양식화대화칸을 열려면 즉 작성하여 표시하기 위하여서는 DialogBox( )라는 API 함수를 호출해야 한다. 아래에 선언을 보여 주었다.

```
int DialogBox(HINSTANCE hThisInst, LPCSTR lpName, HWND hwnd,
                    DLGPROC lpDFunc);
```

hThisInst 는 프로그람의 WinMain( )의 파라메터로서 주어지는 응용프로그람의 실체의 손잡이이다. 자원파일안에 정의되여 있는 대화칸의 이름을 lpName 에 설정한다. 대화칸의 어미창문의 손잡이를 hwnd 에 설정한다.(이것은 일반적으로 DialogBox( )를 호출하는 창문의 손잡이로 된다.) lpDFunc 에는 앞에서 설명한 대화함수의 지시자를 설정한다. 만일 DialogBox( )호출이 실패하면 -1 을 돌려 준다. 그렇지 않은 경우에는 EndDialog( )의 파라메터로서 사용되는 값을 돌려 주는데 이에 대해서는 다음에 설명하기로 한다.

이식과 관련한 요점 : 16bit 의 Windows 에서는 DialogBox( )의 lpDFunc 파라메터가 수속실체(Procedure Instance)를 가리키는 지시자로 되여 있다. 수속실체란 프로그람이 현재 사용하고 있는 자료토막을 사용하여 대화함수에 대한 련결을 진행하는 짧은 프로그람코드이다. 수속실체는 MakeProcInstance( )라는 API 함수를 사용하여 얻는다. 그러나 Windows 2000 에서는 이 수법을 리용하지 않는다. lpDFunc 파라메터에는 대화함수그자체를 설정한다. 낡은 Win16 프로그람코드를 이식하는 경우에는 MakeProcInstance( )를 호출하고 있는 부분을 삭제하여야 한다.

대화칸을 닫기

양식화대화칸을 닫으려면 즉 화면에서 소거하고 파괴하려면 *EndDialog( )*를 사용해야 한다. 아래에 선언을 보여 주었다.

```
BOOL EndDialog( HWND hdwnd, int nStatus);
```

hdwnd 는 대화칸의 손잡이이며 nStatus 는 DialogBox( )함수가 돌려 준 상태코드이다.(특별히 프로그람에서 참조하지 않는한 nStatus 의미에 대하여 의식할 필요는 없다.) 이 함수는 호출이 성공하면 령 아닌 값을 돌려 주며 그렇지 않으면 령을 돌려 준다.

간단한 대화칸의 작성

대화칸의 기본적인 개념을 익히기 위하여 간단한 대화칸을 작성하는것으로부터 시작해 보자. 이 대화칸에는 [Author], [Publisher], [Copyright], [Cancel]의 4 개의 누름단추가 있다.

[Author], [Publisher], [Copyright]중의 하나가 눌리워 지면 이것이 선택되였다는것을 나타내는 대화칸을 표시한다.(후에는 누름단추를 누르면 자료기지정보를 얻게 한다. 이 단계에서는 통보칸만을 표시한다.) [Cancel]단추가 눌리워 진 때는 화면에서 대화칸을 소거한다.

대화칸의 자원파일

대화칸의 설계도 프로그람자원파일안에 정의되여 있는 자원의 일종이다. 대화칸을 사용하는 프로그람을 작성하려면 그것을 정의하는 자원파일이 필요하게 된다.

차림표를 작성할 때와 같이 대화칸의 설계를 본문편집기를 리용하여 서술할수도 있지만 이러한 방법은 그리 쓰이지 않으며 *대화칸편집기*를 사용하는것이 일반적이다. 대화칸의 정의는 내부에 배치되는 여러가지 조종체들의 위치를 결정하는 등의 작업이므로 대화적인 도구를 사용하는 쪽이 보다 효률적이다.

그러나 이 장의 실례프로그람의 자원파일은 모두 본문형식으로 표시되므로 단순히 본문으로 입력한다. 여하튼 프로그람의 대화칸을 작성할 때는 대화칸편집기를 사용하는것이 일반적이다.

대다수의 실제적인 대화칸은 대화칸편집기를 사용하여 작성하므로 이 장의 실례프로그람에서 설명하는것은 자원파일에서의 대화칸정의의 일부분이라고 생각하면 된다.

대화칸은 자원파일에서 DIALOG 또는 DIALOGEX 명령문을 리용하여 정의한다. DIALOG 은 낡은 형식의 명령문이지만 현재도 사용할수 있다. DIALOGEX 에는 편리한 확장기능들이 있으며 다음 장에서도 리용한다. 현재는 DIALOGEX 를 리용하는것을 장려하고 있으므로 이 책에서는 이 명령문을 리용한다.

이 장에서 사용하는 DIALOGEX 명령문은 다음과 같다.

```
Dialog-name DIALOGEX X, Y, WIDTH, HEIGHT
Features
```

```
{
    Dialog-items
}
```

Dialog-name은 대화칸의 이름이다. 대화칸의 왼쪽웃모서리의 자리표를 X, Y에 지정하고 크기를 WIDTH와 HEIGHT에서 지정한다. 대화칸에는 많은 기능들을 지정할수 있다. 뒤에서 설명하지만 이 기능들중에는 대화칸의 이름이나 대화칸형식의 지정 등이 있다. Dialog-items에서는 대화칸에 포함되는 조종체들을 지정한다.

아래의 자원파일은 이 장의 첫 실례프로그람에서 사용되는 자원의 정의이다.

대화칸을 여는데 사용되는 차림표, 차림표의 지름건, 차림표자체가 정의된다. 이 파일을 DIALOG.RC로서 이름을 붙인다.

```
// 대화칸의 실례와 차림표자원파일
#include <windows.h>
#include <dialog.h>
MyMenu MENU
{
    POPUP "&Dialog"
    {
            MENUITEM "&Dialog\tF2", IDM_DIALOG
            MENUITEM "E&xit\tCtrl+X", IDM_EXIT
    }
    MENUITEM "&Help", IDM_HELP
}
MyMenu ACCELERATORS
{
    VK_F2, IDM_DIALOG, VIRTKEY
    "^X", IDM_EXIT
    VK_F1, IDM_HELP, VIRTKEY
}
MyDB DIALOGEX 10, 10, 210, 110
CAPTION "Books Dialog Box"
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION |
WS_SYSMENU
{
    DEFPUSHBUTTON "Author", IDD_AUTHOR, 11, 10, 36, 14
    PUSHBUTTON "Publisher", IDD_PUBLISHER, 11, 34, 36, 14
    PUSHBUTTON "Copyright", IDD_COPYRIGHT, 11, 58, 36, 14
```

```
        PUSHBUTTON "Cancel", IDCANCEL, 11, 82, 36, 16
    }
```

여기에 정의되여 있는 MyDB 라는 이름의 대화칸왼쪽웃모서리의 자리표는 (10, 10)으로 되여 있다. 너비는 210이며 높이는 110이다. *CAPTION*뒤에 놓이는 문자렬은 대화칸의 제목으로 된다. *STYLE* 명령문은 작성하는 대화칸의 형식을 지정한다. 이 장에서 리용되는 형식을 포함하여 일반적인 형식의 설정값을 표 5-1에 주었다. 작성하려는 대화칸에 따라 적절한 형식을 OR로 지정하여 결합할수 있다. 이 형식의 설정값은 조종체에서도 사용할수 있다.

표 5-1. 일반적인 대화칸의 형식

설 정 값	의 미
DS_MODALFRAME	양식화틀거리를 가지는 대화칸
WS_BORDER	경계선을 가진다. 이 형태는 양식화대화칸에서 사용할수 있다.
WS_CAPTION	제목띠를 가진다.
WS_CHILD	새끼창문으로서 작성한다.
WS_POPUP	튀여나오기창문으로서 작성한다.
WS_SYSMENU	체계차림표를 가진다.
WS_TABSTOP	조종체는 타브정지점을 가진다.
WS_VISIBLE	여는것과 동시에 표시된다.

MyDB 의 정의에서 4 개의 누름단추가 정의되였다. 첫 단추는 체계설정의 누름단추로 되여 있다. 이 단추는 대화칸이 처음 표시될 때 자동적으로 강조표시된다. 누름단추를 정의하는 일반적인 구문은 아래와 같다.

PUSHBUTTON "string", PBID, X, Y, Width, Height, [, Style]

string은 누름단추에 표시되는 문자렬이다. PBID는 누름단추를 식별하기 위한 값이다. 단추가 눌리울 때 이 값이 프로그람에 전달된다. 단추의 왼쪽웃모서리의 자리표는 X, Y로 정의되며 단추의 크기는 Width 와 Height 로 정의된다. Style에는 누름단추의 겉모양을 지정한다. 만일 Style 에 아무것도 지정되여 있지 않으면 체계설정값인 *WS_TABSTOP*가 리용되여 단추가 표시된다.

체계설정누름단추를 지정하는 경우에는 *DEFPUSHBUTTON* 문을 사용한다. 이 명령문의 파라메터는 일반적인 단추와 같다.

실례프로그람에서 사용되는 DIALOG.H 머리부파일의 내용을 아래에 표시한다.

```
#define IDM_DIALOG          100
#define IDM_EXIT            101
#define IDM_HELP            102

#define IDD_AUTHOR          200
#define IDD_PUBLISHER       201
#define IDD_COPYRIGHT       202
```

대화칸의 창문함수

대화칸에서 발생한 사건은 프로그람의 기본창문함수가 아니라 대화칸창문함수에 전송된다. 아래에 표시한 대화칸창문함수는 MyDB 라는 대화칸에서 발생한 사건에 응답하기 위한것이다.

```
// 간단한 대화함수
BOOL CALLBACK DialogFunc(HWND hdwnd, UINT message,
                         WPARAM wParam, LPARAM lparam)
{
   switch(message) {
         case WM_COMMAND:
               EndDialog(hdwnd, 0);
               return 1;
         case IDD_COPYRIGHT:
               MessageBox(hdwnd, "Copyright", "Copyright",MB_OK);
               return 1;
         case IDD_AUTHOR:
               MessageBox(hdwnd, "Author", "Author", MB_OK);
               return 1;
         case IDD_PUBLISHER:
               MessageBox(hdwnd, "Publisher", "Publisher", MB_OK);
               return 1;
   }
}
```

대화칸안에 있는 조종체가 조작되면 *DialogFunc( )*에 WM_COMMAND 통보문이 전

송되며 이때 LOWORD(wParam)에 조작된 조종체의 ID 가 보관된다. 조종체의 손잡이는 lParam 에 보관된다.

*DialogFunc( )*에서는 대화칸안에서 발생하는 4 개의 통보문(누름단추의 ID)을 처리한다. 사용자가 [Cancel]을 누르면 *IDCANCEL* 이 발송되므로 EndDialog( )라는 API 함수를 호출하여 대화칸을 닫는다.(IDCANCEL 은 WINDOWS.H 에 정의되여 있는 표준 ID 이다.) 다른 세개의 단추를 누르면 이것들이 선택되였다는것을 표시하는 통보칸이 표시된다. 이미 설명한것처럼 이 단추들은 후에 자료기지의 정보를 표시하는데 쓰인다.

대화칸의 첫 실례프로그람

실례 5-1 에 대화칸의 실례프로그람을 주었다. 이 프로그람을 실행하면 차림표띠에 웃준위차림표가 표시된다. 차림표에서 [Dialog]를 선택하면 대화칸이 표시된다. 대화칸이 표시되면 누름단추를 눌러 그 결과를 확인한다. 이미 서적자료기지가 프로그람코드에 포함되여 있지만 아직 사용할수는 없다. 이것은 다음 실례프로그람에서 필요되는것이다.

실례 5-1. Dialog 프로그람

```
// 양식화대화칸의 실례
#include <windows.h>
#include <cstring>
#include <cstdio>
#include "dialog.h"

#define NUMBOOKS 7

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK DialogFunc(HWND, UINT, WPARAM, LPARAM);

char szWinName[] = "MyWin"; // 창문클라스의 이름

HINSTANCE hInst;

// 서적자료기지
struct booksTag {
  char title[40];
  unsigned copyright;
```

```
  char author[40];
  char publisher[40];
} books[NUMBOOKS] = {
  { "C++: The Complete Reference", 1998,
    "Herbert Schildt", "Osborne/McGraw-Hill" },
  { "MFC Programming from the Ground Up", 1998,
    "Herbert Schildt", "Osborne/McGraw-Hill" },
  { "Java: The Complete Reference", 1999,
    "Naughton and Schildt", "Osborne/McGraw-Hill" },
  { "The C++ Programming Language", 1997,
    "Bjarne Stroustrup", "Addison-Wesley" },
  { "Inside OLE", 1995,
    "Kraig Brockschmidt", "Microsoft Press" },
  { "HTML Sourcebook", 1996,
    "Ian S. Graham", "John Wiley & Sons" },
  { "Standard C++ Library", 1995,
    "P. J. Plauger", "Prentice-Hall" }
};

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;
  HACCEL hAccel;

  // 창문클라스를 정의한다.
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst;      // 실체의 손잡이
  wcl.lpszClassName = szWinName; // 창문클라스의 이름
  wcl.lpfnWndProc = WindowFunc;  // 창문함수
  wcl.style = 0;                 // 체계설정형식

  wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
  wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
  wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

  // 차림표자원의 이름을 지정한다.
```

```
wcl.lpszMenuName = "MyMenu";

wcl.cbClsExtra = 0;       // 보조기억기령역은 불필요함
wcl.cbWndExtra = 0;

// 창문의 배경색을 흰색으로 한다.
wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

// 창문클라스를 등록한다.
if(!RegisterClassEx(&wcl)) return 0;

/* 창문클라스를 등록하였으므로
   창문을 작성할수 있다. */
hwnd = CreateWindow(
  szWinName, // 창문클라스의 이름
  "Demonstrate Dialog Boxes", // 제목
  WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
  CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 너비는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 높이는 Windows가 결정하게 한다.
  NULL,          // 어미창문은 없다.
  NULL,          // 클라스차림표의 덧쓰기는 하지 않는다.
  hThisInst,     // 실체의 손잡이
  NULL           // 추가파라메터는 없다.
);

hInst = hThisInst; // 현재 실체의 손잡이를 보관한다.

// 가속기를 적재한다.
hAccel = LoadAccelerators(hThisInst, "MyMenu");

// 창문을 표시한다.
ShowWindow(hwnd, nWinMode);
UpdateWindow(hwnd);

// 통보문순환고리를 작성한다.
while(GetMessage(&msg, NULL, 0, 0))
{
  if(!TranslateAccelerator(hwnd, hAccel, &msg)) {
    TranslateMessage(&msg); // 건반통보를 변환한다.
```

```
      DispatchMessage(&msg);  // Windows 2000 에 조종을 넘긴다.
    }
  }

  return msg.wParam;
}

/* 이 함수는 Windows 2000 으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받는다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  int response;

  switch(message) {
    case WM_COMMAND:
      switch(LOWORD(wParam)) {
        case IDM_DIALOG:
          DialogBox(hInst, "MyDB", hwnd, (DLGPROC) DialogFunc);
          break;
        case IDM_EXIT:
          response = MessageBox(hwnd, "Quit the Program?",
                                "Exit", MB_YESNO);
          if(response == IDYES) PostQuitMessage(0);
          break;
        case IDM_HELP:
          MessageBox(hwnd, "No Help", "Help", MB_OK);
          break;
      }
      break;
    case WM_DESTROY: // 프로그람을 끝낸다.
      PostQuitMessage(0);
      break;
    default:
      /* 이 switch 문에서 지정된것 이외의 통보문은
         Windows 2000 이 처리하게 한다. */
      return DefWindowProc(hwnd, message, wParam, lParam);
  }
```

```
  return 0;
}

// 간단한 대화함수
BOOL CALLBACK DialogFunc(HWND hdwnd, UINT message,
                         WPARAM wParam, LPARAM lParam)
{
  switch(message) {
    case WM_COMMAND:
      switch(LOWORD(wParam)) {
        case IDCANCEL:
          EndDialog(hdwnd, 0);
          return 1;
        case IDD_COPYRIGHT:
          MessageBox(hdwnd, "Copyright", "Copyright", MB_OK);
          return 1;
        case IDD_AUTHOR:
          MessageBox(hdwnd, "Author", "Author", MB_OK);
          return 1;
        case IDD_PUBLISHER:
          MessageBox(hdwnd, "Publisher", "Publisher", MB_OK);
          return 1;
      }
  }

  return 0;
}
```

프로그람의 실행결과를 그림 5-1에 주었다.

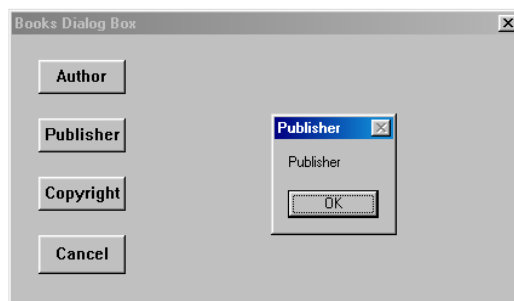



그림 5-1. 대화칸프로그람의 실행결과

대역변수 hInst 를 주목해야 한다. 이 변수는 WinMain( )에 전송되는 실체의 손잡이를 보관하기 위한것이다. 이 변수가 필요하게 되는 리유는 대화칸실체의 손잡이를 참조할 필요가 있기때문이다. 대화칸은 WinMain( )함수내에서 작성되는것이 아니라 WindowFunc( )내에서 작성된다. 그러므로 WinMain( )밖에서도 참조할수 있도록 실체의 손잡이를 변수에 보관하는것이다.

목록칸의 추가

대화칸의 기능을 계속하여 알아보기 위해 앞의 프로그람에서 정의된 대화칸에 다른 조종체를 추가해 보자. 누름단추처럼 자주 사용되는 조종체로서 목록칸이 있다. 여기서는 자료기지의 표제목록을 표시하고 사용자가 흥미를 가지는 항목을 선택할수 있도록 하기 위해 목록칸을 사용한다. *LISTBOX* 문의 일반적인 구문은 아래와 같다.

LISTBOX LBID W, Y ,Width, Height [, Style]

LBID 는 목록칸을 식별하기 위한 값이다. 목록칸의 왼쪽웃모서리의 자리표를 X,Y 에 지정하고 크기를 Width, Height 에 지정한다. Style 에는 목록칸의 외형을 지정한다. 이미 설명한 표준적인 형식의 설정값밖에도 목록칸에는 자체의 형식도 있다.

자주 사용되는 형식을 아래에 주었다.

형　　식	설　　명
LBS_NOTIFY	항목이 두번 찰칵되였다는것을 어미창문에 알려 준다.
LBS_SORT	목록을 분류한다.
LBS_MULTICOLUMN	여러개렬의 목록칸으로 한다.
LBS_EXTENDEDSEL	두개이상의 항목을 선택하는것이 가능하다. 체계설정으로는 동시에 한개 항목밖에 선택할수 없다.

Style 에 아무것도 지정하지 않으면 체계설정으로 LBS_NOTIFY 와 *WS_BORDER* 가 선택되게 되며 목록칸이 표시된다.

목록칸을 추가하기 위해서 DIALOG.RC 에서 대화칸의 정의를 변경한다. 우선 대화칸의 정의에 목록칸을 추가한다.

```
LISTBOX IDD_LB1, 60, 5, 140, 33, LBS_NOTIFY | WS_BORDER |
                    WS_VSCROLL | WS_TABSTOP
```

다음에 아래의 누름단추를 대화칸의 정의에 추가한다.

```
PUSHBUTTON "Select Book" , IDD_SELECT, 103, 41, 54, 14
```

이러한 변경을 진행한 대화칸의 정의는 아래와 같이 된다.

```
MyDB DIALOGEX 10, 10, 210, 110
CAPTION "Books Dialog Box"
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION |
WS_SYSMENU
{
    DEFPUSHBUTTON "Author", IDD_AUTHOR, 11, 10, 36, 14
    PUSHBUTTON "Publisher", IDD_PUBLISHER, 11, 34, 36, 14
    PUSHBUTTON "Copyright", IDD_COPYRIGHT, 11, 58, 36, 14
    PUSHBUTTON "Cancel", IDCANCEL, 11, 82, 36, 16
LISTBOX IDD_LB1, 60, 5, 140, 33, LBS_NOTIFY | WS_BORDER |
                                        WS_VSCROLL | WS_TABSTOP
PUSHBUTTON "Select Book" , IDD_SELECT, 103, 41, 54, 14
}
```

DIALOG.H에는 아래의 마크로를 추가하여야 한다.

```
#define IDD_LB1            203
#define IDD_SELECT         204
```

IDD_LB1 는 자원파일의 대화칸의 정의에서 목록칸을 식별하는 값이다. *IDD_SELECT*는 [Select]누름단추를 식별하는 값이다.

목록칸의 기본적인 사용법

목록칸을 사용할 때는 두가지 처리를 해야 한다. 우선 대화칸이 처음 표시될 때 목록칸을 초기화해야 한다. 이 처리는 표시할 목록항목을 목록칸에 보내는것으로 된다.(체계설정으로 목록칸은 비여 있다.) 목록칸이 초기화되면 사용자가 목록을 선택한것에 응답할 필요가 있다.

목록칸은 여러가지 *통지문*을 생성한다. 통지문은 조종체에서 어떠한 종류의 사건이 발생하였는가를 알려 주는것이다.(표준조종체들은 통지문을 생성한다.)

다음 실례프로그람의 목록칸에서 사용되는 통지문은 *LBN_DBLCLK*이다. 이 통보문은 사용자가 목록칸의 항목을 두번 찰칵할 때 발송된다. 이 통보문은 목록칸이

WM_COMMAND 통보문을 생성할 때 HIWORD(wParam)에 넣어 진다. (LBN_DBLCLK 를 생성하려면 목록칸의 정의에서 Style 에 LBS_NOTIFY 를 포함시켜야 한다.) 선택이 진행된 경우에는 어느 항목이 선택되였는가를 알아 보기 위하여 목록칸에 문의할 필요가 있다.

누름단추와 달리 목록칸은 통보문을 생성할뿐아니라 통보문을 받을수 있는 조종체이다. 목록칸에는 몇가지 종류의 통보문을 보낼수 있다. 목록칸(또는 기타 조종체)에 통보문을 보내기 위하여 *SendDlgItemMessage( )*라는 API 함수를 사용한다. 아래에 함수의 선언을 보여 주었다.

```
LONG SendDlgItemMessage( HWND hdwnd, int ID, UINT IDMsg,
                         WPARAM wParam, LPARAM lParam);
```

*SendDlgItemMessage( )*는 ID 에 식별자를 설정하고 IDMsg 에 설정된 통보문을대화칸안의 조종체에 보낸다. 대화칸의 손잡이는 hdwnd 에 설정한다. 통보문의 부가정보는 wParam, lParam 에 설정한다. 이 부가정보는 통보문의 종류에 따라 다르다. 만일 조종체에 보내는 부가정보가 없다면 wParam 및 lParam에 령을 설정한다. SendDlgItem Message( )의 돌림값은 IDMsg 의 값에 의하여 여러가지 정보를 포함하게 된다.

아래에 목록칸에 보낼수 있는 몇가지 통보문의 마크로를 준다.

마크로	목 적
LB_ADDSTRING	목록칸에 문자렬(선택항목)을 추가한다.
LB_GETCURSEL	선택된 항목의 색인을 요구한다.
LB_SETCURSEL	항목을 선택상태로 한다.
LB_FINDSTRING	문자렬을 검색한다.
LB_SELECTSTRING	문자렬을 검색하고 선택상태로 한다.
LB_GETTEXT	항목에 대응한 문자렬을 얻는다.

이 통보문들의 내용을 상세히 설명해 보자.

*LB_ADDSTRING*은 목록칸에 문자렬을 추가한다. 지정된 문자렬이 목록칸의 새로운 항목으로서 추가된다. 문자렬의 지시자를 lParam 에 설정한다.(이 통보문에서는 wParam 이 사용되지 않는다.) 목록칸이 돌려 주는 값은 목록에 추가된 항목의 색인으로 된다. 오유가 발생한 경우 *LB_ERR*를 돌려 준다.

LB_GETCURSEL 은 현재 선택되여 있는 항목의 색인을 목록칸으로부터 돌려 준다. 목록칸항목의 색인은 령으로부터 시작한다. lParam 과 wParam 은 다 사용되지 않는다. 오유가 발생한 경우 *LB_ERR* 가 돌려 진다. 항목이 선택되여 있지 않은 경우에도 LB_ERR 가 돌려 진다.

LB_SETCURSEL 를 사용하면 목록칸에서 선택상태로 할 항목을 지정할수 있다. 이 통보문에서는 wParam 에 선택상태로 할 항목의 색인을 지정한다. lParam 은 사용되지 않는다. 오유가 발생한 경우에는 *LB_ERR*가 돌려 진다.

LB_FINDSTRING 을 사용하면 목록중의 항목을 검색할수 있다. LB_FINDSTRING 은 지정한 문자렬과 부분적으로 일치하는 목록항목을 검색한다. wParam에는 검색을 개시할 항목의 색인을 설정하고 lParam 에는 검색하려는 문자렬의 지시자를 설정한다. 일치하는 항목을 찾으면 그것의 색인을 돌려 준다. 그렇지 않은 경우에는 *LB_ERR*가 돌려 진다. 또한 LB_FINDSTRING 은 목록칸의 항목을 선택상태로 하지 않는다.

검색된 항목을 선택상태로 하려는 경우에는 LB_SELECTSTRING 을 사용한다. 파라메터는 LB_FINDSTRING 와 같지만 일치하는 항목을 선택상태로 한다.

LB_GETTEXT 를 사용하면 목록칸의 항목의 문자렬을 얻을수 있다. 이 경우에는 wParam에 항목의 색인을 설정하고 lParam에는 얻는 NULL로 종결되는 문자렬을 보관하기 위한 문자렬지시자를 설정한다. 호출이 성공하면 문자렬의 길이를 돌려 주며 실패하면 LB_ERR 가 돌려 진다.

목록칸의 초기화

이미 설명한바와 같이 작성된 직후의 목록칸은 비여 있는 상태이다. 이것은 목록칸이 표시될 때마다 목록칸을 초기화해야 한다는것을 의미한다. 이 처리는 그리 어렵지 않다. 대화칸이 작성될 때마다 WM_INITDIALOG 통보문이 보내지도록 되여 있으므로 DialogFunc( )의 switch 문아래에 case 를 추가하면 목록칸을 초기화할수 있다.

```
case WM_INITDIALOG:                    //목록칸을 초기화한다.
     for(i=0; i<NUMBOOKS; i++)
               SendDlgItemMessage(hdwnd, IDD_LB1,
               LB_ADDSTRING, 0, (LPARAM)books[i].Title);
//   초기항목을 선택상태로 한다.
     SnedDlgItemMessage(hdwnd, IDD_LB1,LB_SETCURSEL, 0, 0);


return 1;
```

이 프로그람코드는 배렬 books에 정의된 서적제목들을 목록칸에 넣는다. 매개 문자렬은 LB_ADDSTRING 통보문을 지정한 SnedDlgItemMessage( )를 반복호출하여 목록칸에 추가된다. 추가하는 문자렬의 지시자는 lParam 에 설정한다.(이때 지시자를 부호없는 옹근수로 하기 위하여 LPARAM 에로의 형변환이 필요하게 된다.) 이 실례프로그람에서는 매개 문자렬이 보내여 지는 순서로 목록칸에 추가된다.(그러나 만일 목록칸의 Style 에 LBS_SORT 가 포함되여 있다면 항목이 영문자모순서로 분류된다.) 목록칸에 보낸 항목의 수가 창문에 표시할수 있는 범위보다 큰 경우에는 자동적으로 수직흘림띠가 추가된다.

이 프로그람코드에서는 목록칸의 첫 항목을 선택상태로 하기 위한 처리도 진행한다. 목록칸이 작성된 직후에는 아무 항목도 선택되지 않은 상태로 되여 있다. 이러한 상태로 해둘 필요가 있는 경우도 있지만 여기에서는 그렇게 하지 않는다. 사용자의 리용상 편리를 고려하여 목록칸의 첫 항목이 선택된 상태로 초기화하여 놓는것이 일반적이다.

***참고** : WM_INITDIALOG 는 목록칸이 열려질 때마다 발송된다. 이 통보문을 받을 때 대화칸에서 필요되는 모든 초기화처리를 진행하여야 한다.*

선택의 처리

목록칸이 초기화되면 그것을 사용할수 있는 준비가 된것으로 된다. 사용자가 목록칸을 선택하는 방법에는 두가지가 있다.

첫번째 방법은 목록칸의 항목을 두번 찰칵하는것이다. 이렇게 하면 WM_COMMAND 통보문이 대화칸의 창문함수에 돌려 진다. 이 경우에는 LOWORD (wParam)에 목록칸의 ID가 보관되며 HIWORD(wParam)에 LB_DBLCLK라는 통지문이 넣어 진다. 두번 찰칵에 의하여 프로그람은 사용자의 선택을 즉시 알수 있다.

두번째 방법은 목록칸을 선택항목을 반전표시시키기 위해서만 사용하는것이다. (한번 찰칵 또는 방향건을 사용하여 반전표시할 항목을 이동할수 있다.) 목록칸은 항목이 선택된 상태를 보관하고 프로그람으로부터의 요구를 기다린다. 실례프로그람에서는 두가지 방법을 다 리용하였다. 목록칸의 항목이 선택되였다면 목록칸에 LB_GETCURSEL 통보문을 보내여 어느 항목이 선택되였는가를 알수 있다. 목록칸은 선택된 항목의 색인을 돌려 준다. 항목이 선택되기전에 이 통보문을 보내면 목록칸은 LB_ERR를 돌려 준다. 이로부터 목록칸이 초기화될 때 어떤 항목을 선택상태로 해놓는것이 좋다는것을 알수 있다.

목록칸의 선택을 처리하기 위하여 DialogFunc( )의 switch 문 아래에 case 를 추가한다. 긴 옹근수 i 및 문자렬 str 를 DialogFunc( )에서 선언해야 한다. 현재 대화칸은 그림 5-2에서 보여 준것과 같다. 두번 찰칵 또는 [Select Book]누름단추가 눌리울 때마다 항목이 선택되며 선택된 서적에 대한 정보가 통보칸에 표시된다.

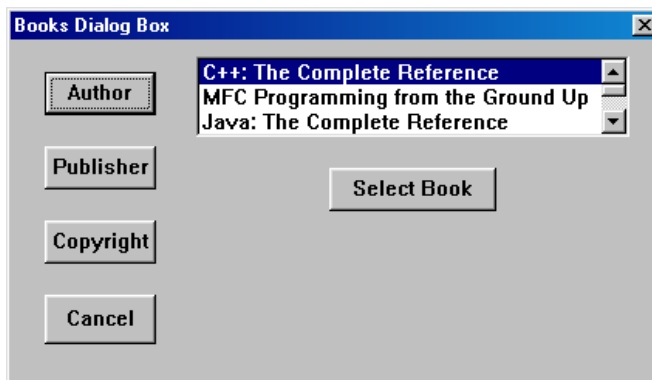



그림 5-2. 목록칸이 있는 대화칸

```
case IDD_LB1: // 목록칸의 LBN_DBLCLK를 처리한다.
    // 사용자가 선택을 진행하였는가를 확인한다.
    if(HIWORD(wParam)==LBN_DBLCLK) {
            i = SendDlgItemMessage(hdwnd, IDD_LB1,
                    LB_GETCURSEL, 0, 0); // 색인을 얻는다.
            sprintf(str, "%s\n%s\n%s, %u",
                    books[i].title, books[i].author,
                    books[i].publisher, books[i].copyright);

            MessageBox(hdwnd, str, "Selection Made", MB_OK);

            // 색인에 대응한 문자렬을 얻는다.
            SendDlgItemMessage(hdwnd, IDD_LB1, LB_GETTEXT,
                i, (LPARAM) str);
    }
    return 1;
case IDD_SELECT: // [Select Book] 단추가 눌리웠다.
        i = SendDlgItemMessage(hdwnd, IDD_LB1,
                LB_GETCURSEL, 0, 0); // 색인을 얻는다.
        sprintf(str, "%s\n%s\n%s, %u",
                books[i].title, books[i].author,
                books[i].publisher, books[i].copyright);

        MessageBox(hdwnd, str, "Selection Made", MB_OK);

        // 색인에 대응한 문자렬을 얻는다.
        SendDlgItemMessage(hdwnd, IDD_LB1, LB_GETTEXT,
            i, (LPARAM) str);

        return 1;
```

IDD_LB1의 case문의 프로그람코드를 주목해야 한다. 목록칸은 여러 종류의 통지문을 생성하므로 사용자가 항목을 두번 찰칵하였다는것을 식별하려면 wParam의 웃단어를 확인해야 한다. 이것은 조종체가 통지문을 생성하여도 그것이 두번 찰칵통보문이라고 단정하지 않는다는것이다.(흥미가 있다면 목록칸의 다른 통지문들도 조사해 볼수 있다.)

편집칸의 추가

여기에서는 대화칸에 편집칸을 추가한다. *편집칸*은 사용자가 임의의 문자렬을 입력할 때 편리하다. 실례프로그람에서는 사용자가 편집칸에서 서적의 제목을 입력하도록 한다.

제목과 일치하는 항목이 목록에 있다면 그것이 선택되고 서적에 대한 정보가 표시된다. 편집칸을 추가하여 간단한 자료기지응용프로그람의 기능을 확장할수 있지만 여기에는 다른 목적도 있다. 그것은 두개의 조종체를 호상 련관속에서 동작시키는 방법을 배우는것이다.

편집칸을 사용하기에 앞서 자원파일에 그의 정의를 추가해야 한다. 여기서는 MyDB를 아래와 같이 변경한다.

```
MyDB DIALOGEX 10, 10, 210, 110
CAPTION "Books Dialog Box"
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION |
WS_SYSMENU
{
    DEFPUSHBUTTON "Author", IDD_AUTHOR, 11, 10, 36, 14
    PUSHBUTTON "Publisher", IDD_PUBLISHER, 11, 34, 36, 14
    PUSHBUTTON "Copyright", IDD_COPYRIGHT, 11, 58, 36, 14
    PUSHBUTTON "Cancel", IDCANCEL, 11, 82, 36, 16
LISTBOX IDD_LB1, 60, 5, 140, 33, LBS_NOTIFY | WS_BORDER |
  WS_VSCROLL | WS_TABSTOP
PUSHBUTTON "Select Book" , IDD_SELECT, 103, 41, 54, 14
EDITTEXT IDD_EB1, 65, 73, 130, 12, ES_LEFT |
                WS_BORDER | ES_AUTHORSCROLL | WS_TABSTOP
PUSHBUTTON "Title Search", IDD_DONE, 107, 91, 46, 14
}
```

사용자가 편집칸에서 서적의 제목을 입력하였다는것을 프로그람에 알려 주기 위해 [Title Search]라는 누름단추를 추가하였다. 추가된 편집칸의 ID는 IDD_EB1이다. 이 정의에서는 표준적인 편집칸이 작성된다.

*EDITTEXT*문의 일반적인 구문을 아래에 주었다.

EDITTEXT EDID, X, Y, Width, Height, [, Style]

EDID는 편집칸을 식별하는 값이다. 편집칸의 왼쪽웃모서리의 자리표는 X, Y에 지

정하고 크기는 Width, Height 에 지정한다. Style 에는 편집칸의 형식을 지정한다. 표 5-1 에 준 형식설정값밖에 EDITTEXT 는 다른 형식들도 제공한다. 그중에서 가장 중요한것은 *ES_AUTOHSCROLL* 일것이다. 이 형식에서는 편집칸의 본문이 좌우로 자동적으로 이동된다.

다음 DIALOG.H 의 정의에 아래의 마크로를 추가한다.

```
#define IDD_EB1          205
#define IDD_DONE         206
```

편집칸은 많은 통보문을 받으며 또한 여러가지 통보문들을 생성한다. 그러나 이 실례프로그람의 목적에 따르면 프로그람에서 응답해야 하는 통보문은 없다. 편집칸은 내부적으로 편집기능을 갖추고 있으며 프로그람에서 처리할 필요는 없다. 프로그람에서 처리해야 하는것은 편집칸의 현재 내용을 얻는 시간을 결정하는것뿐이다.

편집칸의 현재 내용을 얻으려면 *GetDlgItemText( )* 라는 API 함수를 사용해야 한다. 아래에 선언을 보여 주었다.

```
UINT GetDlgItemText(HWND hdwnd, int nID, LPSTR lpstr, int nMax);
```

이 함수는 편집칸의 내용을 lpstr 이 가리키는 문자렬에 복사한다. 대화칸의 손잡이는 hdwnd 에 설정한다. 편집칸의 ID 는 nID 에 지정한다. 문자렬을 복사하는 최대길이는 nMax 에 지정한다. 이 함수는 실제로 얻은 문자렬의 길이를 돌려 준다.

모든 응용프로그람에서 필요되는것은 아니지만 *SetDlgItemText( )*라는 함수를 사용하여 편집칸의 내용을 초기화할수 있다. 아래에 선언을 보여 주었다.

```
BOOL SetDlgItemText(HWND hdwnd, int nID, LPSTR lpstr);
```

이 함수는 lpstr 가 가리키는 문자렬을 편집칸의 내용으로 보관한다. 대화칸의 손잡이는 hdwnd 에 지정한다. 편집칸의 ID 는 nID 에 지정한다. 함수호출이 성공하면 령이 아닌 값을 돌려 주며 실패하면 령을 돌려 준다.

실례프로그람에 편집칸을 추가하려면 아래의 case 문을 DialogFunc( )의 switch 문에 추가한다. [Title Search]단추가 눌리울 때마다 편집칸의 현재 문자렬과 일치하는 제목이 목록칸의 항목으로부터 검색된다. 검색이 일치한 경우에는 그 제목이 목록칸에서 선택된다.

제목은 앞의 몇문자를 입력하여도 관계 없다. 목록칸은 부분적인 문자렬과 일치하는 제목의 검색을 자동적으로 진행한다.

```
    case IDD_DONE: // [Title Search] 단추가 눌리웠다.
      // 편집칸의 현재 내용을 얻는다.
```

```
GetDlgItemText(hdwnd, IDD_EB1, str, 80);

// 목록칸에서 일치하는 문자렬을 검색한다.
i = SendDlgItemMessage(hdwnd, IDD_LB1, LB_FINDSTRING,
        0, (LPARAM) str);

if(i != LB_ERR) { // 검색이 일치한 경우
  // 일치한 제목을 목록칸에서 선택한다.
  SendDlgItemMessage(hdwnd, IDD_LB1, LB_SETCURSEL, i, 0);

  // 색인에 대응한 문자렬을 얻는다.
  SendDlgItemMessage(hdwnd, IDD_LB1, LB_GETTEXT,
      i, (LPARAM) str);

  // 색인의 문자렬을 갱신한다.
  SetDlgItemText(hdwnd, IDD_EB1, str);
}
else
  MessageBox(hdwnd, str, "No Title Matching", MB_OK);
return 1;
```

이 프로그람코드에서는 편집칸의 현재내용을 얻고 그것과 일치하는 문자렬을 편집칸에서 검색한다. 검색이 일치하였다면 목록칸의 항목을 선택하고 목록칸의 문자렬을 편집칸에 복사한다. 이렇게 두개의 조종체가 호상 련관속에서 동작하게 된다.

INITDIALOG 의 case 문에는 다음의 프로그람코드도 추가해야 한다. 이것은 대화칸이 표시될 때마다 편집칸을 초기화하기 위한것이다.

```
// 편집칸을 초기화한다.
SetDlgItemText(hdwnd, IDD_EB1, books[0].title);
```

이러한 변경들외에도 목록칸의 처리를 진행하는 프로그람코드도 선택된 서적의 제목을 자동적으로 편집칸에 복사하도록 개조하여야 한다. 이러한 개조는 다음에 보여 주는 완성된 프로그람에 반영되게 된다.

양식화대화칸의 완성된 프로그람

누름단추, 목록칸, 편집칸이 있는 양식화대화칸의 완성된 실례프로그람을 실례 5-2에 주었다. 누름단추의 처리를 진행하는 프로그람코드는 목록칸에 현재 선택되여 있는

제목과 관련한 정보를 표시하도록 되여 있다. 그림 5-3 에는 이 프로그람의 실행결과를 주었다.

실례 5-2. ModalDialog 프로그람

```
// 양식화대화칸의 완전한 프로그람코드
#include <windows.h>
#include <cstring>
#include <cstdio>
#include "dialog.h"

#define NUMBOOKS 7

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK DialogFunc(HWND, UINT, WPARAM, LPARAM);

char szWinName[] = "MyWin"; // 창문클라스의 이름

HINSTANCE hInst;

// 서적자료기지
struct booksTag {
  char title[40];
  unsigned copyright;
  char author[40];
  char publisher[40];
} books[NUMBOOKS] = {
  { "C++: The Complete Reference", 1998,
    "Herbert Schildt", "Osborne/McGraw-Hill" },
  { "MFC Programming from the Ground Up", 1998,
    "Herbert Schildt", "Osborne/McGraw-Hill" },
  { "Java: The Complete Reference", 1999,
    "Naughton and Schildt", "Osborne/McGraw-Hill" },
  { "The C++ Programming Language", 1997,
    "Bjarne Stroustrup", "Addison-Wesley" },
  { "Inside OLE", 1995,
    "Kraig Brockschmidt", "Microsoft Press" },
  { "HTML Sourcebook", 1996,
```

```
    "Ian S. Graham", "John Wiley & Sons" },
  { "Standard C++ Library", 1995,
    "P. J. Plauger", "Prentice-Hall" }
};

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;
  HACCEL hAccel;

  // 창문클라스를 정의한다.
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst;     // 실체의 손잡이
  wcl.lpszClassName = szWinName; // 창문클라스의 이름
  wcl.lpfnWndProc = WindowFunc;  // 창문함수
  wcl.style = 0;                 // 체계설정의 형식

  wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
  wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
  wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

  // 차림표자원의 이름을 지정한다.
  wcl.lpszMenuName = "MyMenu";

  wcl.cbClsExtra = 0;      // 보조기억기령역은 불필요함
  wcl.cbWndExtra = 0;

  // 창문의 배경색은 흰색으로 한다.
  wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

  // 창문클라스를 등록한다.
  if(!RegisterClassEx(&wcl)) return 0;

  /* 창문클라스가 등록되였으므로
```

```
       창문을 작성할수 있다. */
    hwnd = CreateWindow(
      szWinName, // 창문클라스의 이름
      "Demonstrate Dialog Boxes", // 제목
      WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
      CW_USEDEFAULT, // X 자리표는 Windows 가 결정하게 한다.
      CW_USEDEFAULT, // Y 자리표는 Windows 가 결정하게 한다.
      CW_USEDEFAULT, // 너비는 Windows 가 결정하게 한다.
      CW_USEDEFAULT, // 높이는 Windows 가 결정하게 한다.
      NULL,          // 어미창문은 없다.
      NULL,          // 클라스차림표의 덧쓰기는 하지 않는다.
      hThisInst,     // 실체의 손잡이
      NULL           // 추가파라메터는 없다.
    );

    hInst = hThisInst; // 현재 실체의 손잡이를 보관한다.

    // 가속기를 적재한다.
    hAccel = LoadAccelerators(hThisInst, "MyMenu");

    // 창문을 표시한다.
    ShowWindow(hwnd, nWinMode);
    UpdateWindow(hwnd);

    // 통보문순환고리를 작성한다.
    while(GetMessage(&msg, NULL, 0, 0))
    {
      if(!TranslateAccelerator(hwnd, hAccel, &msg)) {
        TranslateMessage(&msg); // 건반통보를 변환한다.
        DispatchMessage(&msg);  // Windows 2000 에 조종을 넘긴다.
      }
    }

    return msg.wParam;
  }

  /* 이 함수는 Windows 2000 으로부터 호출되여
     통보문대기렬에서 꺼낸 통보문을 받는다.
```

```
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  int response;

  switch(message) {
    case WM_COMMAND:
      switch(LOWORD(wParam)) {
        case IDM_DIALOG:
          DialogBox(hInst, "MyDB", hwnd, (DLGPROC) DialogFunc);
          break;
        case IDM_EXIT:
          response = MessageBox(hwnd, "Quit the Program?",
                                "Exit", MB_YESNO);
          if(response == IDYES) PostQuitMessage(0);
          break;
        case IDM_HELP:
          MessageBox(hwnd, "No Help", "Help", MB_OK);
          break;
      }
      break;
    case WM_DESTROY: // 프로그람을 끝낸다.
      PostQuitMessage(0);
      break;
    default:
      /* 이 switch 문에서 지정되지 않은 통보문은
         Windows 2000 에 처리를 맡긴다. */
      return DefWindowProc(hwnd, message, wParam, lParam);
  }

  return 0;
}

// 간단한 대화함수
BOOL CALLBACK DialogFunc(HWND hdwnd, UINT message,
                         WPARAM wParam, LPARAM lParam)
{
```

```
long i;
char str[255];

switch(message) {
  case WM_COMMAND:
    switch(LOWORD(wParam)) {
      case IDCANCEL:
        EndDialog(hdwnd, 0);
        return 1;
      case IDD_COPYRIGHT:
        i = SendDlgItemMessage(hdwnd, IDD_LB1,
                LB_GETCURSEL, 0, 0);  // 색인을 얻는다.
        sprintf(str, "%u", books[i].copyright);
        MessageBox(hdwnd, str, "Copyright", MB_OK);
        return 1;
      case IDD_AUTHOR:
        i = SendDlgItemMessage(hdwnd, IDD_LB1,
                LB_GETCURSEL, 0, 0);  // 색인을 얻는다.
        sprintf(str, "%s", books[i].author);
        MessageBox(hdwnd, str, "Author", MB_OK);
        return 1;
      case IDD_PUBLISHER:
        i = SendDlgItemMessage(hdwnd, IDD_LB1,
                LB_GETCURSEL, 0, 0);  // 색인을 얻는다.
        sprintf(str, "%s", books[i].publisher);
        MessageBox(hdwnd, str, "Publisher", MB_OK);
        return 1;
      case IDD_DONE: // [Title Search] 단추가 눌리웠다.
        // 편집칸의 현재 내용을 얻는다.
        GetDlgItemText(hdwnd, IDD_EB1, str, 80);

        // 목록칸의 문자렬을 검색한다.
        i = SendDlgItemMessage(hdwnd, IDD_LB1, LB_FINDSTRING,
                0, (LPARAM) str);

        if(i != LB_ERR) { // 검색이 일치한 경우
          // 목록칸의 항목을 선택한다.
          SendDlgItemMessage(hdwnd, IDD_LB1, LB_SETCURSEL, i, 0);
```

```
        // 색인에 대응한 문자렬을 얻는다.
        SendDlgItemMessage(hdwnd, IDD_LB1, LB_GETTEXT,
            i, (LPARAM) str);

        // 편집칸의 문자렬을 갱신한다.
        SetDlgItemText(hdwnd, IDD_EB1, str);
      }
      else
        MessageBox(hdwnd, str, "No Title Matching", MB_OK);
      return 1;
    case IDD_LB1: // 목록칸의 LBN_DBLCLK 을 처리한다.
      // 사용자가 선택을 진행하였는가를 확인한다.
      if(HIWORD(wParam)==LBN_DBLCLK) {
        i = SendDlgItemMessage(hdwnd, IDD_LB1,
                LB_GETCURSEL, 0, 0); // 색인을 얻는다.
        sprintf(str, "%s\n%s\n%s, %u",
                books[i].title, books[i].author,
                books[i].publisher, books[i].copyright);

        MessageBox(hdwnd, str, "Selection Made", MB_OK);

        // 색인에 대응한 문자렬을 얻는다.
        SendDlgItemMessage(hdwnd, IDD_LB1, LB_GETTEXT,
            i, (LPARAM) str);

        // 편집칸을 갱신한다.
        SetDlgItemText(hdwnd, IDD_EB1, str);
      }
      return 1;
     case IDD_SELECT: // [Select Book] 단추가 눌리웠다.
      i = SendDlgItemMessage(hdwnd, IDD_LB1,
              LB_GETCURSEL, 0, 0); // 색인을 얻는다.
      sprintf(str, "%s\n%s\n%s, %u",
              books[i].title, books[i].author,
              books[i].publisher, books[i].copyright);

      MessageBox(hdwnd, str, "Selection Made", MB_OK);

      // 색인에 대응한 문자렬을 얻는다.
      SendDlgItemMessage(hdwnd, IDD_LB1, LB_GETTEXT,
          i, (LPARAM) str);
```

```
            // 편집칸을 갱신한다.
            SetDlgItemText(hdwnd, IDD_EB1, str);
            return 1;
        }
        break;
      case WM_INITDIALOG: // 목록칸을 초기화한다.
        for(i=0; i<NUMBOOKS; i++)
          SendDlgItemMessage(hdwnd, IDD_LB1,
                    LB_ADDSTRING, 0, (LPARAM)books[i].title);

        // 첫 항목을 선택상태로 한다.
        SendDlgItemMessage(hdwnd, IDD_LB1, LB_SETCURSEL, 0, 0);

        // 편집칸을 초기화한다.
        SetDlgItemText(hdwnd, IDD_EB1, books[0].title);

        return 1;
    }

    return 0;
}
```

이 프로그람의 실행결과를 그림 5-3에 주었다.

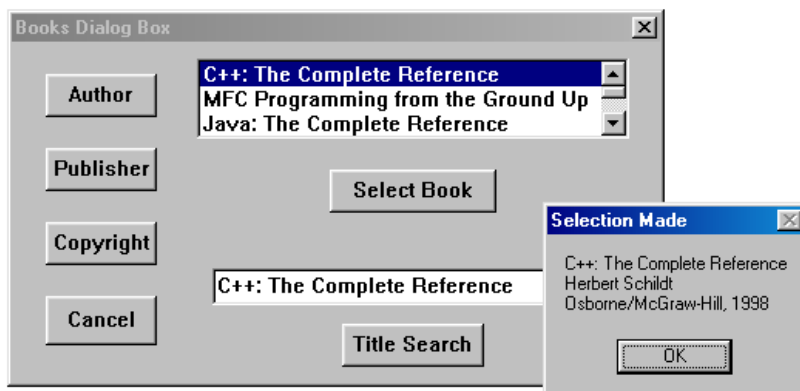



그림 5-3. 양식화대화칸의 완성된 프로그람실행결과

비양식화대화칸의 사용법

이 장을 끝내기에 앞서 지금까지 프로그람에서 사용한 양식화대화칸을 *비양식화대화칸*으로 개조해 보자. 비양식화대화칸을 리용하자면 약간한 추가작업이 필요하게 된다. 그의 주되는 리유는 비양식화대화칸이 양식화대화칸에 비하여 보다 독립성이 높기때문이다. 비양식화대화칸이 표시될 때도 프로그람의 다른 부분은 능동인 상태 그대로 있다. 비양식화대화칸의 창문함수와 응용프로그람의 창문함수는 둘 다 통보문을 받아 들인다. 그러므로 비양식화대화칸을 리용하려면 응용프로그람의 통보문순환고리에도 기능을 추가하여야 한다. 비양식화대화칸을 작성하는데는 DialogBox( )를 리용할수 없다. 대신에 *CreateDialog( )* 라는 API 함수를 사용하여야 한다. 아래에 이 함수의 선언을 보여 주었다.

```
HWND CreateDialog(HINSTANCE hThisInst, LPCSTR lpName,
                  HWND hwnd, DLGPROC lpDFunc);
```

hThisInst 는 프로그람의 WinMain( )의 파라메터로서 주어 진 실체의 손잡이이다. 자원파일에서 정의된 대화칸의 이름을 lpName 에 설정한다. 대화칸의 어미창문의 손잡이를 hwnd 에 설정한다.(이것은 일반적으로 CreateDialog( )를 호출하는 창문의 손잡이이다.)

*lpDFunc( )*에는 대화함수의 손잡이를 설정한다. 이 대화함수는 양식화대화칸에서 리용되던것과 동일한 형식으로 되여 있다. CreateDialog( )는 대화칸의 손잡이를 돌려준다. 대화칸을 작성할수 없는 경우에는 NULL 을 돌려 준다.

양식화대화칸과 달리 비양식화대화칸은 자동적으로 표시되지 않는다. 비양식화대화칸을 작성한후에 ShowWindow( )함수를 호출하여 표시하여야 한다. 그러나 자원파일의 대화칸정의에 *WS_VISIBLE* 을 추가하면 자동적으로 표시된다. 비양식화대화칸을 닫자면 EndDialog( )가 아니라 *DestroyWindow( )*함수를 호출하여야 한다. 아래에 이 함수의 선언을 주었다.

```
BOOL DestroyWindow(HWND hwnd);
```

hwnd 는 닫으려는 창문(이 경우에는 대화칸)의 손잡이이다. 이 함수의 돌림값은 호출이 성공하면 령이 아닌 값이며 실패하면 령이다.

응용프로그람의 창문함수는 비양식화대화칸이 능동인 상태라고 해도 통보문을 계속 받아 들이고 있으므로 프로그람의 통보문순환고리에 *IsDialogMessage( )*함수의 호출을 추가하여야 한다. 이 함수는 대화칸의 통보문을 비양식화대화칸에 보낸다. 아래에 선언을 보여 주었다.

BOOL IsDialogMessage(HWND hdwnd, LPMSG msg);

hdwnd 는 비양식화대화칸의 손잡이이며 msg 는 프로그람의 통보문순환고리의 GetMessage( )함수에서 받아 들이는 통보문이다. 이 함수의 돌림값은 통보문이 대화칸에 전송되는것이라면 TRUE 로 된다. 그렇지 않은 경우에는 FALSE 로 된다. 통보문이 대화칸에 보내는것인 경우에는 그것이 대화칸의 창문함수에 자동적으로 전달된다. 그리하여 hDlg를 대화칸의 손잡이로 하여 대화칸의 통보문을 처리한다면 프로그람의 통보문순환고리는 아래와 같이 된다.

```
while(GetMessage(&msg, NULL, 0, 0))
{
  if(!hDlg || !IsDialogMessage(hDlg, &msg)) {
    // 대화칸의 통보문이 아닌 경우
    if(!TranslateAccelerator(hwnd, hAccel, &msg)) {
      TranslateMessage(&msg); // 전반통보를 변환한다.
      DispatchMessage(&msg);  // Windows 2000 에 조종을 넘긴다.
    }
  }
}
```

이것을 보면 알수 있는바와 같이 hDlg 가 NULL 이거나 통보문이 대화칸에 보내는것이 아닌 경우에만 통보문순환고리의 나머지 부분의 처리가 진행된다. hDlg 의 값을 검사하여 IsDialogMessage( )가 무효한 손잡이를 호출하는것을 방지할수 있다.

다시 한보 전진

조종체를 무효로 하기

조종체를 임의의 상황에서 항시 사용가능하게는 하지 말아야 할 경우도 있다. 만일 필요하다면 조종체를 무효로 할수 있다. 조종체를 유효 혹은 무효로 하려면 *EnableWindow( )*라는 API 함수를 사용한다. 아래에 그 선언을 보여 주었다.

BOOL EnableWindow(HWND hCntl, BOOL How);

hCntl 은 대상으로 되는 창문의 손잡이이다.(조종체도 창문의 일종이다.) How 가 령이 아닌 경우에는 조종체가 유효로 되고 사용가능하게 된다. How 가 령인 경우에는 조종체가 무효로 된다. 조종체가 이미 무효로 되여 있던 경우에 함수의 돌림값은 령이 아닌 값이고 조종체가 직전까지 유효로 되여 있던 경우에

함수의 돌림값은 령으로 된다. 조종체의 손잡이를 받기 위하여 *GetDlgItem( )* 이라는 API 함수를 쓴다. 아래에 선언을 보여 주었다.

```
HWND GetDlgItem(HWND, inr ID);
```

hDwnd는 조종체가 속한 대화칸의 손잡이이다. 조종체의 식별자를 ID에 설정한다. 이 값은 자원파일에서 조종체에 지정한것이다. 함수의 돌림값은 지정된 조종체의 손잡이로 되며 호출이 실패한 경우에는 NULL로 된다.

이 함수들을 사용하여 조종체를 무효로 하는 방법을 보여 주는 실례를 보자. 아래의 프로그람코드는 [Author]누름단추를 무효로 하는것이다. 여기서 hwpb는 HWND형의 변수이다.

```
hwpb = GetDlgItem( hdwnd, IDD_AUTHOR); // 단추의 손잡이를 얻는다.
EnableWindows(hwpb, 0); //단추를 무효로 한다.
```

이 장과 다음 장의 실례프로그람에서 사용되는 다른 조종체들을 무효나 유효로 하는것은 독자들에게 맡긴다.

비양식화대화칸의 작성

지금까지 작성한 실례프로그람의 양식화대화칸을 비양식화대화칸으로 개조하는데 필요한 변경개소는 놀랄만큼 적다.

우선 DIALOG.RC 자원파일에서 대화칸의 정의를 변경한다. 비양식화대화칸은 자동적으로 표시되지 않으므로 대화칸의 정의에 WS_VISIBLE을 추가한다. *DS_MODALFRAME*을 삭제하고 이것을 WS_BORDER로 바꾸어놓는다. 이런 변경을 진행한 DIALOG.RC의 내용은 다음과 같다.

```
// 대화칸의 실례와 차림표의 자원파일
#include <windows.h>
#include <dialog.h>
MyMenu MENU
{
    POPUP "&Dialog"
    {
            MENUITEM "&Dialog\tF2", IDM_DIALOG
            MENUITEM "E&xit\tCtrl+X", IDM_EXIT
    }
```

```
        MENUITEM "&Help", IDM_HELP
    }

    MyMenu ACCELERATORS
    {
        VK_F2, IDM_DIALOG, VIRTKEY
        "^X", IDM_EXIT
        VK_F1, IDM_HELP, VIRTKEY
    }
    MyDB DIALOGEX 10, 10, 210, 110
    CAPTION "Books Dialog Box"
    STYLE WS_POPUP | WS_CAPTION | WS_SYSMENU | WS_VISIBLE
                | WS_BORDER
    {
        DEFPUSHBUTTON "Author", IDD_AUTHOR, 11, 10, 36, 14
        PUSHBUTTON "Publisher", IDD_PUBLISHER, 11, 34, 36, 14
        PUSHBUTTON "Copyright", IDD_COPYRIGHT, 11, 58, 36, 14
        PUSHBUTTON "Cancel", IDCANCEL, 11, 82, 36, 16
        LISTBOX IDD_LB1, 60, 5, 140, 33, LBS_NOTIFY | WS_BORDER |
                        WS_VSCROLL | WS_TABSTOP
        PUSHBUTTON "Select Book" , IDD_SELECT, 103, 41, 54, 14
        EDITTEXT IDD_EB1, 65, 73, 130, 12, ES_LEFT |
                        WS_BORDER | ES_AUTHORSCROLL | WS_TABSTOP
        PUSHBUTTON "Title Search", IDD_DONE, 107, 91, 46, 14
    }
```

이 프로그람을 다음과 같이 변경한다.

- hDlg 라는 대역변수를 추가하고 NULL 로 초기화한다.
- 통보문순환고리에 IsDialogMessage( )를 추가한다.
- DialogBox( )가 아니라 CreateDialog( )로 대화칸을 작성한다.
- EndDialog( )가 아니라 DestroyWindow( )로 대화칸을 닫는다.

비양식화대화칸의 완성된 실례프로그람을 실례 5-3 에 주었다. hDlg 에 특히 주목해야 한다. 이 변수는 NULL 로 초기화된다. 대화칸이 작성되면 그의 손잡이가 hDlg 에 넣어 진다. 대화칸을 닫으면 hDlg 는 NULL 로 재설정된다. 따라서 hDlg 는 대화칸이 능동상태인 때만 NULL 이 아닌 값으로 된다. hDlg 는 통보문순환고리에서 검사되므로

무효한 손잡이로 IsDialogMessage( )가 호출되는 일은 없다.

프로그람의 실행결과를 그림 5-4 에 주었다.(이 프로그람을 보면 양식화대화칸과 비양식화대화칸과의 차이를 알수 있다.)

실례 5-3. ModelessDialog 프로그람

```
// 비양식화대화칸의 실례

#include <windows.h>
#include <cstring>
#include <cstdio>
#include "dialog.h"

#define NUMBOOKS 7

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK DialogFunc(HWND, UINT, WPARAM, LPARAM);

char szWinName[] = "MyWin"; // 창문클라스의 이름

HINSTANCE hInst;

HWND hDlg = 0; // 대화칸의 손잡이

// 서적자료기지
struct booksTag {
  char title[40];
  unsigned copyright;
  char author[40];
  char publisher[40];
} books[NUMBOOKS] = {
  { "C++: The Complete Reference", 1998,
    "Herbert Schildt", "Osborne/McGraw-Hill" },
  { "MFC Programming from the Ground Up", 1998,
    "Herbert Schildt", "Osborne/McGraw-Hill" },
  { "Java: The Complete Reference", 1999,
    "Naughton and Schildt", "Osborne/McGraw-Hill" },
  { "The C++ Programming Language", 1997,
```

```
      "Bjarne Stroustrup", "Addison-Wesley" },
    { "Inside OLE", 1995,
      "Kraig Brockschmidt", "Microsoft Press" },
    { "HTML Sourcebook", 1996,
      "Ian S. Graham", "John Wiley & Sons" },
    { "Standard C++ Library", 1995,
      "P. J. Plauger", "Prentice-Hall" }
};

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;
  HACCEL hAccel;

  // 창문클라스를 정의한다.
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst;       // 실체의 손잡이
  wcl.lpszClassName = szWinName; // 창문클라스의 이름
  wcl.lpfnWndProc = WindowFunc;  // 창문함수
  wcl.style = 0;                   // 체계설정의 형식

  wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
  wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
  wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

  // 차림표자원의 이름을 지정한다.
  wcl.lpszMenuName = "MyMenu";

  wcl.cbClsExtra = 0;       // 보조기억기령역은 불필요함
  wcl.cbWndExtra = 0;

  // 창문의 배경색은 흰색으로 한다.
  wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

  // 창문클라스를 등록한다.
  if(!RegisterClassEx(&wcl)) return 0;

  /* 창문클라스가 등록되였으므로
```

```
    창문을 작성할수 있다. */
  hwnd = CreateWindow(
    szWinName, // 창문클라스의 이름
    "Demonstrate a Modeless Dialog Box", // 제목
    WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
    CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
    CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
    CW_USEDEFAULT, // 너비는 Windows가 결정하게 한다.
    CW_USEDEFAULT, // 높이는 Windows가 결정하게 한다.
    NULL,          // 어미창문은 없다.
    NULL,          // 클라스차림표의 덧쓰기는 하지 않는다.
    hThisInst,     // 실체의 손잡이
    NULL           // 추가파라메터는 없다.
  );

  hInst = hThisInst; // 현재 실체의 손잡이를 보관한다.

  // 가속기를 적재한다.
  hAccel = LoadAccelerators(hThisInst, "MyMenu");

  // 창문을 표시한다.
  ShowWindow(hwnd, nWinMode);
  UpdateWindow(hwnd);

  // 통보문순환고리를 작성한다.
  while(GetMessage(&msg, NULL, 0, 0))
  {
    if(!hDlg || !IsDialogMessage(hDlg, &msg)) {
      // 대화칸통보문이 아닌 경우
      if(!TranslateAccelerator(hwnd, hAccel, &msg)) {
        TranslateMessage(&msg); // 건반통보를 변환한다.
        DispatchMessage(&msg);  // Windows 2000에 조종을 넘긴다.
      }
    }
  }

  return msg.wParam;
}

/* 이 함수는 Windows 2000으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받는다.
```

```
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  int response;

  switch(message) {
    case WM_COMMAND:
      switch(LOWORD(wParam)) {
        case IDM_DIALOG:
          hDlg = CreateDialog(hInst, "MyDB", hwnd,
                         (DLGPROC) DialogFunc);
          break;
        case IDM_EXIT:
          response = MessageBox(hwnd, "Quit the Program?",
                              "Exit", MB_YESNO);
          if(response == IDYES) PostQuitMessage(0);
          break;
        case IDM_HELP:
          MessageBox(hwnd, "No Help", "Help", MB_OK);
          break;
      }
      break;
    case WM_DESTROY: // 프로그람을 끝낸다.
      PostQuitMessage(0);
      break;
    default:
      /* 이 switch 문에서 지정된것 이외의 통보문은
         Windows 2000 에 처리를 맡긴다. */
      return DefWindowProc(hwnd, message, wParam, lParam);
  }

  return 0;
}

// 간단한 비양식화대화함수
BOOL CALLBACK DialogFunc(HWND hdwnd, UINT message,
                         WPARAM wParam, LPARAM lParam)
```

```
long i;
char str[255];

switch(message) {
  case WM_COMMAND:
    switch(LOWORD(wParam)) {
      case IDCANCEL:
        DestroyWindow(hdwnd);
        hDlg = 0; // 손잡이를 무효로 한다.
        return 1;
      case IDD_COPYRIGHT:
        i = SendDlgItemMessage(hdwnd, IDD_LB1,
                LB_GETCURSEL, 0, 0); // 색인을 얻는다.
        sprintf(str, "%u", books[i].copyright);
        MessageBox(hdwnd, str, "Copyright", MB_OK);
        return 1;
      case IDD_AUTHOR:
        i = SendDlgItemMessage(hdwnd, IDD_LB1,
                LB_GETCURSEL, 0, 0); // 색인을 얻는다.
        sprintf(str, "%s", books[i].author);
        MessageBox(hdwnd, str, "Author", MB_OK);
        return 1;
      case IDD_PUBLISHER:
        i = SendDlgItemMessage(hdwnd, IDD_LB1,
                LB_GETCURSEL, 0, 0); // 색인을 얻는다.
        sprintf(str, "%s", books[i].publisher);
        MessageBox(hdwnd, str, "Publisher", MB_OK);
        return 1;
      case IDD_DONE: // [Title Search] 단추가 눌리웠다.
        // 편집칸의 현재 내용을 얻는다.
        GetDlgItemText(hdwnd, IDD_EB1, str, 80);

        // 목록칸의 문자렬을 검색한다.
        i = SendDlgItemMessage(hdwnd, IDD_LB1, LB_FINDSTRING,
                0, (LPARAM) str);

        if(i != LB_ERR) { // 검색이 일치한 경우
          // 목록칸의 항목을 선택한다.
          SendDlgItemMessage(hdwnd, IDD_LB1, LB_SETCURSEL, i, 0);
```

```
        // 색인에 대응한 문자렬을 얻는다.
        SendDlgItemMessage(hdwnd, IDD_LB1, LB_GETTEXT,
            i, (LPARAM) str);

        // 색인에 대응한 문자렬을 얻는다.
        SetDlgItemText(hdwnd, IDD_EB1, str);
      }
      else
        MessageBox(hdwnd, str, "No Title Matching", MB_OK);
      return 1;
    case IDD_LB1: // 목록칸의 LBN_DBLCLK를 처리한다.
      // 사용자가 선택을 진행하였는가를 확인한다.
      if(HIWORD(wParam)==LBN_DBLCLK) {
        i = SendDlgItemMessage(hdwnd, IDD_LB1,
                LB_GETCURSEL, 0, 0); // 색인을 얻는다.
        sprintf(str, "%s\n%s\n%s, %u",
                books[i].title, books[i].author,
                books[i].publisher, books[i].copyright);

        MessageBox(hdwnd, str, "Selection Made", MB_OK);

        // 색인에 대응한 문자렬을 얻는다.
        SendDlgItemMessage(hdwnd, IDD_LB1, LB_GETTEXT,
            i, (LPARAM) str);

        // 편집칸을 갱신한다.
        SetDlgItemText(hdwnd, IDD_EB1, str);
      }
      return 1;
     case IDD_SELECT: // [Select Book]단추가 눌리웠다.
      i = SendDlgItemMessage(hdwnd, IDD_LB1,
              LB_GETCURSEL, 0, 0); // 색인을 얻는다.
      sprintf(str, "%s\n%s\n%s, %u",
              books[i].title, books[i].author,
              books[i].publisher, books[i].copyright);

      MessageBox(hdwnd, str, "Selection Made", MB_OK);

      // 색인에 대응한 문자렬을 얻는다.
      SendDlgItemMessage(hdwnd, IDD_LB1, LB_GETTEXT,
          i, (LPARAM) str);
```

```
            // 편집칸을 갱신한다.
            SetDlgItemText(hdwnd, IDD_EB1, str);
            return 1;
        }
        break;
      case WM_INITDIALOG: // 목록칸을 초기화한다.
        for(i=0; i<NUMBOOKS; i++)
          SendDlgItemMessage(hdwnd, IDD_LB1,
                  LB_ADDSTRING, 0, (LPARAM)books[i].title);

        // 첫 항목을 선택상태로 한다.
        SendDlgItemMessage(hdwnd, IDD_LB1, LB_SETCURSEL, 0, 0);

        // 편집칸을 초기화한다.
        SetDlgItemText(hdwnd, IDD_EB1, books[0].title);

        return 1;
    }

    return 0;
}
```

그림 5-4. 비양식화대화칸의 실레프로그람의 실행결과

제 6 장

여러가지 조종체

이 장에서는 Windows 2000 의 표준적인 조종체들중에서 세개의 새 조종체들을 취급한다. 이것들은 흘림띠, 검사칸, 단일선택단추이다. 5 장에서 학습한 조종체의 사용법을 이 장에서 배우는 조종체에도 적용할수 있다.

구체적으로는 먼저 흘림띠에 대한 설명으로부터 시작하여 간단한 실례프로그람을 작성해본다. 흘림띠는 다른 조종체들보다 취급하기 시끄럽지만 그다지 어려운것은 아니다.

다음에 검사칸과 단일선택단추에 대하여 학습한다. 흘림띠, 검사칸, 단일선택단추의 기본적인 사용법을 보여 주기 위하여 간단한 내려세기시계 프로그람을 작성한다. 이 프로그람을 작성할 때 시계의 새치기와 *WM_TIMER* 통보문에 대한 설명을 한다. 이 장에서는 Windows 의 정적조종체에 대하여서도 학습한다.

흘 림 띠

*흘림띠*는 Windows 의 조종체들중에서 가장 중요한것의 하나로서 두가지 종류가 있다. 첫번째 종류는 완전히 창문이나 대화칸의 일부로 되는것인데 이것을 *표준흘림띠*라고 한다. 두번째 종류는 조종체로서 독립적으로 존재하는것인데 이젓을 *흘림띠조종체*라고 한다. 어느 종류의 흘림띠라고 해도 거의나 같은 방법으로 취급할수 있다.

창문에 표준흘림띠의 추가

창문에 표준흘림띠를 추가하려면 몇가지 설정을 해야 한다. 응용프로그람의 기본창문에서 CreateWindow()를 사용하여 창문을 작성할 때 창문의 형식을 설정하는 파라메터에 *WS_VSCROLL* 이나 *WS_HSCROLL* 을 지정하면 표준흘림띠가 추가된다.

대화칸인 경우에는 자원파일에서 대화칸의 형식을 정의할 때 WS_VSCROLL 나 WS_HSCROLL 을 지정한다. 말그대로 WS_VSCROLL 은 수직흘림띠, WS_HSCROLL 은 수평흘림띠를 추가한다. 이러한 형식들을 지정하면 수직흘림띠나 수평흘림띠가 자동적으로 창문에 추가된다.

흘림띠통보문의 처리

다른 조종체와는 달리 표준흘림띠나 흘림띠조종체는 WM_COMMAND 통보문을 생성하지 않는다. 그대신 수직흘림띠나 수평흘림띠를 조작하면 WS_VSCROLL 이나 WS_HSCROLL 통보문이 생성된다. 이때 wParam 의 아래단어의 값은 조작내용을 가리킨다. 표준흘림띠에서 lParam 은 령이다. 그러나 흘림띠조종체가 생성한 통보문인 경우에는 lParam 에 조종체의 손잡이가 보관된다. LOWORD(wParam)의 값은 흘림띠에 어떠한 조작이 가해졌는가를 가리킨다. 주되는 값은 아래와 같다.

SB_LINEUP	SB_LINERIGHT
SB_LINEDOWN	SB_PAGELEFT
SB_PAGEDOWN	SB_PAGERIGHT
SB_PAGEUP	SB_THUMBPOSITION
SB_LINELEFT	SB_THUMBTRACK

수직흘림띠에서는 사용자가 수직흘림띠를 웃방향으로 움직이면 *SB_LINEUP* 이 발송되며 흘림띠를 아래방향으로 이동하면 *SB_LINEDOWN* 이 발송된다. *SB_PAGEUP*, *SB_PAGEDOWN* 은 페지단위로 흘림띠를 아래우로 움직였을 때 발송된다.

수평흘림띠에서는 사용자가 흘림띠를 왼쪽으로 이동하면 *SB_LINELEFT*가 발송된다. 흘림띠를 오른쪽으로 이동하면 *SB_LINERIGHT* 가 발송된다. *SB_PAGELEFT*, *SB_PAGERIGHT*는 페지단위로 흘림띠를 좌우로 움직였을 때 발송된다.

어떤 형태의 흘림띠에서든지 *SB_THUMBPOSITION*은 흘림띠의 *흘림칸*이 새로운 위치에 이동된 다음에 발송된다. 또한 *SB_THUMBTRACK* 도 흘림칸이 새로운 위치에 이동될 때 발송된다. 이것들의 차이는 SB_THUMBTRACK 는 흘림칸이 새로운 위치에로 이동하는 도중에 발송된다는것이다. 이렇게 하여 흘림칸을 놓기전에 이동위치를 추적할 수 있다.

SB_THUMBPOSITION 또는 SB_THUMBTRACK 를 받았을 때 wParam 의 웃단어에는 현재 흘림칸의 위치가 보관되여 있다.

이식과 관련한 요점 : 16bit 판 Windows 에서는 여기에서 설명하는 wParam 과 lParam 의 역할이 Windows 2000 과 다르다. 즉 흘림띠의 손잡이는 lParam 의 웃단어에, 흘림칸의 위치는 Iparam 의 아래단어에 그리고 흘림띠의 조작내용은 wParam 에 보관되여 있다. 이러한 차이점이 있으므로 16bit 판 Windows 의 프로그람코드를 Windows 2000 에 이식하려면 흘림띠통보문을 처리하는 부분을 모두 바꾸어 써넣어야 한다.

SetScrollInfo( ) 와 GetScrollInfo( )

흘림띠는 많은 부분을 수동적으로 조작하여야 하는 조종체이다. 이것은 흘림띠통보문의 처리뿐아니라 프로그람코드에서 흘림띠와 관련되는 여러가지 속성들을 설정해야 한다는것을 의미한다. 례를 들면 프로그람코드에서 *흘림칸*의 위치를 설정해 주어야 한다.

Windows 2000 은 흘림띠를 조작하기 위한 두개의 함수를 제공한다. *SetScrollInfo( )*함수는 흘림띠와 관련되는 다양한 속성들의 설정을 진행한다. 아래에 선언을 보여 주었다.

```
int SetScrollInfo( HWND hwnd, int which, LPSCROLLINFO lpSI,
                   BOOL repain);
```

hwnd 는 흘림띠를 식별하기 위한 손잡이이다. 창문에 추가된 표준흘림띠에서는 이 손잡이에 흘림띠가 포함된 창문의 손잡이를 설정한다. 흘림띠조종체에서는 흘림띠자체의 손잡이를 설정한다.

which 의 값은 대상으로 되는 흘림띠를 지정한다. 창문에 추가된 수직흘림띠의 속성을 지정하는 경우는 이 파라메터에 *SB_VERT* 를 설정한다. 창문에 추가된 수평흘림띠의 속성을 설정하는 경우는 *SB_HORZ* 를 설정한다. 흘림띠조종체의 경우에는 이 파라메터에 *SB_CTL*을 설정하고 hwnd 에 조종체의 손잡이를 설정한다.

속성에 대한 정보는 lpSI 에 설정한다.(뒤에서 상세한 설명을 준다.) repaint 파라메터가 TRUE 인 경우에는 흘림띠가 다시그리기되고 FALSE 인 경우에는 다시그리기되

지 않는다. 이 함수는 흘림칸의 위치를 돌려 준다. 흘림띠와 관련한 속성을 얻기 위해서는 *GetScrollInfo( )*를 사용한다. 아래에 선언을 보여 주었다.

BOOL GetScrollInfo(HWND hwnd, int which, LPSCROLLINFO lpSI);

hwnd 와 which 의 의미는 SetScrollInfo( )에서 설명한것과 같다. GetScrollInfo( )에서 얻는 정보는 lpSI 가 가리키는 지시자에 보관된다. 호출이 성공하면 함수는 령이 아닌 값을 돌려 주며 실패하면 령을 돌려 준다.

어느 함수에서나 lpSI 의 자료형은 *SCROLLINFO* 구조체형으로 된다. 아래에 그의 정의를 표시한다.

```
typedef struct tagSCROLLINFO
{
    UINT cbSize;      // SCROLLINFO 구조체의 크기
    UINT fMask;       // 조작내용
    int nMin;         // 최소값
    int nMax;         // 최대값
    UINT nPage        // 페지의 값
    int nPos;         // 흘림칸의 위치
    int nTrackPos;    // 탐색중의 현재위치
}   SCROLLINFO;
```

cbSize 에는 SCROLLINFO 구조체의 크기를 설정한다. fMask 에 설정하거나 보관되여 있는 값은 나머지 파라메터에서 유효한것이 어느것인가를 가리킨다. SetScrollInfo( )를 호출하는 경우에는 흘림띠의 어느 속성을 설정하는가를 fMask 에서 지정한다. GetScrollInfo( )를 호출하는 경우에는 어느 속성을 얻는가를 fMask 에서 지정한다. fMask 의 값은 아래에 준 값 또는 그것들을 조합한 값이여야 한다.(값을 조합하는 경우에는 OR 연산자를 사용한다.)

값	설　　명
SIF_ALL	SIF_PAGE|SIF_POS|SIF_RANGE| SIF_TRACKPOS 와 동일하다.
SIF_DISABLENOSCROLL	범위가 령이여도 흘림띠를 삭제하지 않고 무효로 한다.
SIF_PAGE	nPage 에 유효한 값이 설정되여 있다.
SIF_POS	nPos 에 유효한 값이 설정되여 있다.
SIF_RANGE	nMin 과 nMax 에 유효한 값이 설정되여 있다.
SIF_TRACKPOS	nTrackPos 에 유효한 값이 설정되여 있다.

nPage 는 흘림띠의 크기에 비례한 페지의 값을 표시한다. nPos 는 흘림칸의 위치를 표시한다. nMin 및 nMax 는 흘림띠의 범위의 최소값 및 최대값을 가리킨다. nTrackPos 에는 탐색중의 현재위치가 설정된다. 탐색중의 현재위치란 탐색중의 흘림칸의 현재위치이다. 이 값은 참조만 가능하다.

이식과 관련한 요점 : Win16 프로그람에서는 흘림띠의 설정을 진행하는데 SetScrollRange( ) 와 SetScrollPos( )라는 API 함수를 사용한다. 이 함수들은 Win32 에서도 지원되고 있으나 프로그람을 이식할 때는 SetScrollInfo( )로 변경할것을 권고한다.

흘림띠의 사용방법

앞에서 설명한바와 같이 흘림띠는 수동으로 조작하는 조종체이다. 이것은 흘림띠가 이동되였을 때 흘림칸의 위치를 프로그람코드에서 설정해야 한다는것을 의미한다. 이것을 실현하려면 nPos 에 새로운 위치를 설정하고 fMask 에 *SIF_POS* 를 설정하여 SetScrollInfo( )를 호출한다. 례를 들어 수직흘림띠의 위치를 갱신하기 위해서는 다음과 같이 프로그람코드를 서술한다.

```
SCROLLINFO si;
//...
si.cbSize = sizeof(SCROLLINFO);
si.fMask = SIF_POS;
si.nPos = newposition;
SetScrollInfo(hwnd, SB_VERT, &si, 1);
```

흘림띠의 범위는 흘림띠의 끝에서 끝까지 얼마만한 자리가 있는가를 결정한다. 창문에 추가되는 표준흘림띠의 범위는 체계설정값으로 0~100 으로 되여 있다. 이 값들은 프로그람의 목적에 따라 변경시킬수 있다.

흘림띠조종체의 범위는 체계설정값으로서 0~0 으로 되여 있다. 그러므로 흘림띠조종체를 사용하기에 앞서 범위를 설정하여야 한다.(범위가 령으로 설정되여 있는 흘림띠는 동작하지 않는다.) 범위를 설정하면 프로그람에서 흘림칸의 위치를 쉽게 관리할수 있게 된다.

흘림띠의 실례프로그람

이제부터 작성하게 되는 프로그람은 수직 및 수평 흘림띠를 관리하는 실례프로그람이다. 이 프로그람은 아래의 자원파일을 필요로 한다.

```
// 흘림띠의 실례
#include "scroll.h"
#include <windows.h>
MyMenu MENU
{
     POPUP "&Dialog"
     {
               MENUITEM "&Scroll Bars\F2", IDM_DIALOG
               MENUITEM "E&xit\Ctrl+X", IDM_EXIT
     }
     MENUITEM "&Help", IDM_HELP
}
MyMenu ACCELERATORS
{
     VK_F2, IDM_DIALOG, VIRTKEY
     "^X", IDM_EXIT
     VK_F1, IDM_HELP, VIRTKEY
}
MyDB DIALOGEX 18, 18, 142, 92
CAPTION "Using Scroll Bars"
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION |
WS_SYSMENU | WS_VSCROLL | WS_HSCROLL
{
}
```

보는바와 같이 현재는 대화칸의 정의가 비여 있다. 형식에 WS_VSCROLL 과 WS_HSCROLL 을 지정하면 대화칸에 흘림띠가 자동적으로 추가된다.

아래와 같은 내용의 SCROLL.H 라는 머리부파일도 필요하다.

```
#define IDM_DIALOG          100
#define IDM_EXIT            101
#define IDM_HELP            102
```

실례 6-1 에 흘림띠의 적용실례를 보여 주는 완전한 프로그람코드를 주었다. 수직흘림띠는 흘림칸을 움직일 때 *SB_LINEUP*, *SB_LINEDOWN*, *SB_PAGEUP*, *SB_PAGEDOWN*, *SB_THUMBPOSITION* 및 *SB_ THUMBTRACK* 의 각 통보문에 응답한다. 흘림칸의 현재

위치도 표시된다.(수평흘림띠가 다른 통보문들에도 응답하게 하려면 필요한 프로그람코드를 추가하여야 한다.) 흘림칸을 움직이면 위치의 표시가 변화된다.

수평흘림띠는 *SB_LINELEFT* 및 *SB_LINERIGHT* 에만 응답한다. 흘림칸의 위치도 표시된다.(수평흘림띠가 다른 통보문들에도 응답하게 하려면 필요한 프로그람코드를 추가하여야 한다.)

수직흘림띠 및 수평흘림띠의 범위는 대화칸이 WM_INITDIALOG 통보문을 받았을 때 설정된다는 점에 주목해야 한다. 흘림띠의 범위를 변경하고 그 결과를 확인할수도 있다.

또 하나의 중요한 점은 매 흘림띠의 흘림칸의 위치가 TextOut( )를 리용하여 대화칸의 의뢰자구역에 표시된다는것이다. 대화칸은 특정한 용도에 사용되는것이지만 기본창문과 같은 기능을 가지는 창문이라는데는 다른 점이 없다. 프로그람의 실행결과를 그림 6-1 에 보여 주었다.

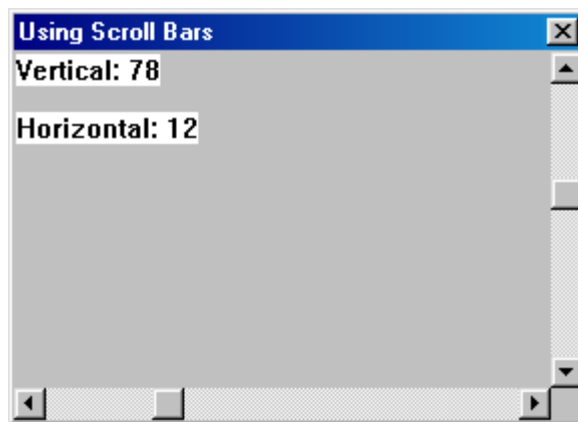



그림 6-1. 표준흘림띠 실례프로그람의 실행결과

실례 6-1. Scroll 프로그람

```
// 표준흘림띠의 실례

#include <windows.h>
#include <cstring>
#include <cstdio>
#include "scroll.h"

#define VERTRANGEMAX 200
#define HORZRANGEMAX  50
```

```
LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK DialogFunc(HWND, UINT, WPARAM, LPARAM);

char szWinName[] = "MyWin"; // 창문클라스의 이름

HINSTANCE hInst;

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;
  HACCEL hAccel;

  // 창문클라스를 정의한다.
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst;      // 실체의 손잡이
  wcl.lpszClassName = szWinName; // 창문클라스의 이름
  wcl.lpfnWndProc = WindowFunc;  // 창문함수
  wcl.style = 0;                 // 체계설정의 형식

  wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
  wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
  wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

  // 차림표자원의 이름을 지정한다.
  wcl.lpszMenuName = "MyMenu";

  wcl.cbClsExtra = 0;      // 보조기억기령역은 불필요함
  wcl.cbWndExtra = 0;

  // 창문의 배경색은 흰색으로 한다.
  wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

  // 창문클라스를 등록한다.
  if(!RegisterClassEx(&wcl)) return 0;
```

```
    /* 창문클라스가 등록되였으므로
      창문을 작성할수 있다. */
    hwnd = CreateWindow(
      szWinName, // 창문클라스의 이름
      "Managing Scroll Bars", // 제목
      WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
      CW_USEDEFAULT, // X 자리표는 Windows 가 결정하게 한다.
      CW_USEDEFAULT, // Y 자리표는 Windows 가 결정하게 한다.
      CW_USEDEFAULT, // 너비는 Windows 가 결정하게 한다.
      CW_USEDEFAULT, // 높이는 Windows 가 결정하게 한다.
      NULL,          // 어미창문은 없음
      NULL,          // 클라스차림표의 덧쓰기는 하지 않는다.
      hThisInst,     // 실체의 손잡이
      NULL           // 추가파라메터는 없음
    );

    hInst = hThisInst; // 현재 실체의 손잡이를 보관한다.

    // 가속기를 적재한다.
    hAccel = LoadAccelerators(hThisInst, "MyMenu");

    // 창문을 표시한다.
    ShowWindow(hwnd, nWinMode);
    UpdateWindow(hwnd);

    // 통보문순환고리를 작성한다.
    while(GetMessage(&msg, NULL, 0, 0))
    {
      if(!TranslateAccelerator(hwnd, hAccel, &msg)) {
        TranslateMessage(&msg); // 건반통보를 변환한다.
        DispatchMessage(&msg);  // Windows 2000 에 조종을 넘긴다.
      }
    }

    return msg.wParam;
  }
```

```
/* 이 함수는 Windows 2000 으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받는다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  int response;

  switch(message) {
    case WM_COMMAND:
      switch(LOWORD(wParam)) {
        case IDM_DIALOG:
          DialogBox(hInst, "MyDB", hwnd, (DLGPROC) DialogFunc);
          break;
        case IDM_EXIT:
          response = MessageBox(hwnd, "Quit the Program?",
                                "Exit", MB_YESNO);
          if(response == IDYES) PostQuitMessage(0);
          break;
        case IDM_HELP:
          MessageBox(hwnd, "Try the Scroll Bar", "Help", MB_OK);
          break;
      }
      break;
    case WM_DESTROY: // 프로그람을 끝낸다.
      PostQuitMessage(0);
      break;
    default:
      /* 이 switch 문에서 지정되지 않은 통보문들은
         Windows 2000 이 처리하게 한다. */
      return DefWindowProc(hwnd, message, wParam, lParam);
  }

  return 0;
}

// 대화함수
BOOL CALLBACK DialogFunc(HWND hdwnd, UINT message,
```

```
                              WPARAM wParam, LPARAM lParam)
{
  char str[80];
  static int vpos = 0;  // 수직흘림띠의 흘림칸의 위치
  static int hpos = 0;  // 수평흘림띠의 흘림칸의 위치
  static SCROLLINFO si; // 흘림띠의 정보를 보관하는 구조체

  HDC hdc;
  PAINTSTRUCT paintstruct;

  switch(message) {
    case WM_COMMAND:
      switch(LOWORD(wParam)) {
        case IDCANCEL:
          EndDialog(hdwnd, 0);
          return 1;
      }
      break;
    case WM_INITDIALOG:
      si.cbSize = sizeof(SCROLLINFO);
      si.fMask = SIF_RANGE;
      si.nMin = 0; si.nMax = VERTRANGEMAX;
      SetScrollInfo(hdwnd, SB_VERT, &si, 1);
      si.nMax = HORZRANGEMAX;
      SetScrollInfo(hdwnd, SB_HORZ, &si, 1);
      vpos = hpos = 0;
      return 1;
    case WM_PAINT:
      hdc = BeginPaint(hdwnd, &paintstruct);
      sprintf(str, "Vertical: %d", vpos);
      TextOut(hdc, 1, 1, str, strlen(str));
      sprintf(str, "Horizontal: %d", hpos);
      TextOut(hdc, 1, 30, str, strlen(str));
      EndPaint(hdwnd, &paintstruct);
      return 1;
    case WM_VSCROLL:
      switch(LOWORD(wParam)) {
        case SB_LINEDOWN:
```

```
          vpos++;
          if(vpos>>VERTRANGEMAX) vpos = VERTRANGEMAX;
          break;
        case SB_LINEUP:
          vpos--;
          if(vpos<0) vpos = 0;
          break;
        case SB_THUMBPOSITION:
          vpos = HIWORD(wParam); // 현재위치를 얻는다.
          break;
        case SB_THUMBTRACK:
          vpos = HIWORD(wParam); // 현재위치를 얻는다.
          break;
        case SB_PAGEDOWN:
          vpos += 5;
          if(vpos>>VERTRANGEMAX) vpos = VERTRANGEMAX;
          break;
        case SB_PAGEUP:
          vpos -= 5;
          if(vpos<0) vpos = 0;
      }
      // 수직흘림띠의 위치를 갱신한다.
      si.fMask = SIF_POS;
      si.nPos = vpos;
      SetScrollInfo(hdwnd, SB_VERT, &si, 1);
      hdc = GetDC(hdwnd);
      sprintf(str, "Vertical: %d    ", vpos);
      TextOut(hdc, 1, 1, str, strlen(str));
      ReleaseDC(hdwnd, hdc);
      return 1;
    case WM_HSCROLL:
      switch(LOWORD(wParam)) {
        /* 여기에 수평흘림띠의 다른 통보문들을 처리하는
                프로그람코드를 추가하여 본다. */
        case SB_LINERIGHT:
          hpos++;
          if(hpos>>HORZRANGEMAX) hpos = HORZRANGEMAX;
          break;
```

```
            case SB_LINELEFT:
              hpos--;
              if(hpos<0) hpos = 0;
          }
          // 수평흘림띠의 위치를 갱신한다.
          si.fMask = SIF_POS;
          si.nPos = hpos;
          SetScrollInfo(hdwnd, SB_HORZ, &si, 1);
          hdc = GetDC(hdwnd);
          sprintf(str, "Horizontal: %d    ", hpos);
          TextOut(hdc, 1, 30, str, strlen(str));
          ReleaseDC(hdwnd, hdc);
          return 1;
      }

      return 0;
  }
```

흘림띠조종체의 사용방법

*흘림띠조종체*는 독자적인 흘림띠이며 창문에 추가된것이 아니다. 흘림띠조종체는 표준흘림띠와 거의 같은 방법으로 조작할수 있지만 두가지 큰 차이점이 있다.

첫번째 차이는 흘림띠조종체의 범위가 체계설정으로서 령으로 되여 있다는것이며 그것을 반드시 설정해야 한다는것이다. 그러므로 초기에는 동작할수 없는 상태로 되여 있다. 이것은 체계설정의 범위가 0∼100 으로 되여 있는 표준흘림띠와 다르다. 두번째 차이점은 흘림띠의 통보문을 받았을 때 lParam 에 보관되여 있는 값의 의미이다.

표준흘림띠와 흘림띠조종체는 모두 흘림띠가 수평 또는 수직인가에 따라 *WM_HSCROLL* 통보문 혹은 *WM_VSCROLL* 통보문을 받아 들인다는것을 상기해 볼 필요가 있다. 이 통보문들을 표준흘림띠가 생성한 경우에 lParam 의 값은 항상 령이다. 이와는 달리 흘림띠조종체가 통보문을 생성한 경우는 lParam 에 조종체의 손잡이가 보관되여 있다. 표준흘림띠와 흘림띠조종체를 다 가지고 있는 창문에서는 어느 흘림띠가 통보문을 생성하였는가를 구별해야 할 필요가 있다.

흘림띠조종체의 작성

흘림띠조종체를 작성하려면 자원파일에서 *SCROLLBAR* 명령문을 사용한다. 아래에

일반적인 서식을 보여 주었다.

SCROLLBAR SBID, X, Y, Width, Height[, Style]

SBID는 흘림띠를 식별하는 값이다. 흘림띠의 왼쪽웃모서리의 자리표를 X, Y에 설정하고 흘림띠의 크기를 Width 및 Height에 설정한다. 흘림띠의 형식은 Style에 설정한다. 체계설정형식은 *SBS_HORZ*이며 이에 의해 수평흘림띠가 작성된다. 건입력초점을 받는 흘림띠로 하려는 경우는 형식에 *WS_TABSTOP*를 포함시킨다.

흘림띠조종체의 실례

흘림띠조종체의 실례를 보여 주기 위해 앞에서 작성한 프로그람을 부분적으로 개조해보자. 우선 다음과 같이 대화칸의 정의를 변경한다. 여기서는 수직흘림띠를 추가해본다.

```
MyDB DIALOGEX 18, 18, 142, 92
CAPTION "Adding a Control Scroll Bar"
STYLE DS MODALFRAME | WS_POPUP | WS_CAPTION |
    WS_SYSMENU | WS_VSCROLL | WS_HSCROLL
{
    SCROLLBAR ID_SB1, 110, 10, 10, 70, SBS_VERT | WS_TABSTOP
}
```

그리고 SCROLL.H에 다음의 한 행을 추가한다.

```
#define ID_SB1 200
```

다음 흘림띠조종체를 조작하기 위한 프로그람코드를 추가한다. 이 프로그람코드는 WM_VSCROLL통보문을 생성하는 표준흘림띠와 흘림띠조종체를 구별한다.

흘림띠조종체가 lParam에 손잡이를 보관한다는것을 고려하여야 한다. 표준흘림띠에서 lParam의 값은 령이다. 례를 들어 아래의 SB_LINEDOWN의 case문은 표준흘림띠와 흘림띠조종체를 구별한다.

```
case SB_LINEDOWN:
    if ((HWND)lParam==GetDlgItem(hdwnd, ID_SB1)) {
            // 흘림띠조종체이다.
            cntlpos++;
            if(cntlpos>VERTRANGEMAX) cntlpos = VERTRANGEMAX;
            }
```

```
                else {
                        // 표준흘림띠이다.
                        vpos++;
                        if(vpos>>VERTRANGEMAX) vpos = VERTRANGEMAX;
                }
                break;
```

이 프로그람코드에서는 lParam 에 보관된 손잡이가 *GetDlgItem( )*에서 얻은 흘림띠조종체의 손잡이와 비교되고 있다. 손잡이의 값이 같다면 통보문이 흘림띠조종체에 의해 생성되였다는것을 알수 있다. 그렇지 않은 경우에 통보문은 표준흘림띠에 의해 생성되였다는것이다.

5 장에서 설명한것처럼 *GetDlgItem( )*이라는 API 는 조종체의 손잡이를 돌려 준다. 다시 한번 선언을 보여 주었다.

HWND GetDlgItem(HWND hDwnd, int ID);

hDwnd는 조종체가 속한 대화칸의 손잡이이다. 조종체를 식별하는 값은 ID에 설정된다. 이것은 자원파일에서 조종체에 지정한 값이다. 이 함수는 지정된 조종체의 손잡이를 돌려 주며 함수의 호출이 실패한 경우는 NULL 을 돌려 준다.

실례 6-2 에 흘림띠조종체를 취급하는 실례프로그람을 보여 주었다. 이것은 몇가지 점을 내놓고는 앞에서 작성한 프로그람코드와 같다. DialogFunc( )에서 흘림띠조종체의 위치를 보관하기 위한 변수 cntlpos가 선언된다. 흘림띠조종체는 WM_INITDIALOG 에서 초기화된다. WM_VSCROLL 통보문을 처리하는 부분에서는 통보문이 표준흘림띠와 흘림띠조종체중 어느것에서 보낸것인가를 판정한다. 흘림띠조종체의 현재위치를 표시하기 위한 프로그람코드도 추가되였다.

실례 6-2. ScrollControl 프로그람

```
// 흘림띠조종체의 실례프로그람

#include <windows.h>
#include <cstring>
#include <cstdio>
#include "scroll.h"

#define VERTRANGEMAX 200
#define HORZRANGEMAX  50

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK DialogFunc(HWND, UINT, WPARAM, LPARAM);
```

```
char szWinName[] = "MyWin"; // 창문클라스의 이름

HINSTANCE hInst;

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;
  HACCEL hAccel;

  // 창문클라스를 정의한다.
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst;     // 실체의 손잡이
  wcl.lpszClassName = szWinName; // 창문클라스의 이름
  wcl.lpfnWndProc = WindowFunc;  // 창문함수
  wcl.style = 0;                 // 체계설정형식

  wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
  wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
  wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

  // 차림표자원의 이름을 지정한다.
  wcl.lpszMenuName = "MyMenu";

  wcl.cbClsExtra = 0;      // 보조기억기령역은 불필요함
  wcl.cbWndExtra = 0;

  // 창문의 배경색을 흰색으로 한다.
  wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

  // 창문클라스를 등록한다.
  if(!RegisterClassEx(&wcl)) return 0;

  /* 창문클라스가 등록되였으므로
     창문을 작성할수 있다. */
  hwnd = CreateWindow(
    szWinName, // 창문클라스의 이름
    "Managing Scroll Bars", // 제목
    WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
    CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
```

```
    CW_USEDEFAULT, // Y 자리표는 Windows가 결정하게 한다.
    CW_USEDEFAULT, // 창문의 너비는 Windows가 결정하게 한다.
    CW_USEDEFAULT, // 창문의 높이는 Windows가 결정하게 한다.
    NULL,          // 어미창문은 없다.
    NULL,          // 클라스차림표의 덧쓰기는 하지 않는다.
    hThisInst,     // 실체의 손잡이
    NULL           // 추가파라메터는 없음
  );

  hInst = hThisInst; // 현재 실체의 손잡이를 보관한다.

  // 가속기를 적재한다.
  hAccel = LoadAccelerators(hThisInst, "MyMenu");

  // 창문을 표시한다.
  ShowWindow(hwnd, nWinMode);
  UpdateWindow(hwnd);

  // 통보문순환고리를 작성한다.
  while(GetMessage(&msg, NULL, 0, 0))
  {
    if(!TranslateAccelerator(hwnd, hAccel, &msg)) {
      TranslateMessage(&msg); // 건반통보를 변환한다.
      DispatchMessage(&msg);  // Windows 2000에 조종을 넘긴다.
    }
  }

  return msg.wParam;
}

/* 이 함수는 Windows 2000으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받는다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  int response;
```

```
  switch(message) {
    case WM_COMMAND:
      switch(LOWORD(wParam)) {
        case IDM_DIALOG:
          DialogBox(hInst, "MyDB", hwnd, (DLGPROC) DialogFunc);
          break;
        case IDM_EXIT:
          response = MessageBox(hwnd, "Quit the Program?",
                                "Exit", MB_YESNO);
          if(response == IDYES) PostQuitMessage(0);
          break;
        case IDM_HELP:
          MessageBox(hwnd, "Try the Scroll Bar", "Help", MB_OK);
          break;
      }
      break;
    case WM_DESTROY: // 프로그람을 끝낸다.
      PostQuitMessage(0);
      break;
    default:
      /* 이 switch 문에서 지정된것 이외의 통보문은
         Windows 2000 에 처리를 맡긴다. */
      return DefWindowProc(hwnd, message, wParam, lParam);
  }

  return 0;
}

// 대화함수
BOOL CALLBACK DialogFunc(HWND hdwnd, UINT message,
                         WPARAM wParam, LPARAM lParam)
{
  char str[80];
  static int vpos = 0;    // 수직흘림띠의 흘림칸의 위치
  static int hpos = 0;    // 수평흘림띠의 흘림칸의 위치
  static int cntlpos = 0; // 흘림띠조종체의 흘림칸의 위치
  static SCROLLINFO si;   // 흘림띠정보를 보관하는 구조체
```

```
    HDC hdc;
    PAINTSTRUCT paintstruct;

    switch(message) {
      case WM_COMMAND:
        switch(LOWORD(wParam)) {
          case IDCANCEL:
            EndDialog(hdwnd, 0);
            return 1;
        }
        break;
      case WM_INITDIALOG:
        si.cbSize = sizeof(SCROLLINFO);
        si.fMask = SIF_RANGE;
        si.nMin = 0; si.nMax = VERTRANGEMAX;

        // 표준수직흘림띠의 위치를 설정한다.
        SetScrollInfo(hdwnd, SB_VERT, &si, 1);

        // 흘림띠조종체의 범위를 설정한다.
        SetScrollInfo(GetDlgItem(hdwnd, ID_SB1), SB_CTL, &si, 1);

        si.nMax = HORZRANGEMAX;
        // 표준수평흘림띠의 범위를 설정한다.
        SetScrollInfo(hdwnd, SB_HORZ, &si, 1);

        vpos = hpos = cntlpos = 0;
        return 1;
      case WM_PAINT:
        hdc = BeginPaint(hdwnd, &paintstruct);
        sprintf(str, "Vertical: %d", vpos);
        TextOut(hdc, 1, 1, str, strlen(str));
        sprintf(str, "Horizontal: %d", hpos);
        TextOut(hdc, 1, 30, str, strlen(str));
        sprintf(str, "Scroll Bar Control: %d    ", cntlpos);
        TextOut(hdc, 1, 60, str, strlen(str));
        EndPaint(hdwnd, &paintstruct);
        return 1;
```

```
case WM_VSCROLL:
  /* 여기에서는 통보문을 생성한것이
      흘림띠조종체인가 표준흘림띠인가를 판정해야 한다. */
  switch(LOWORD(wParam)) {
    case SB_LINEDOWN:
      if((HWND)lParam==GetDlgItem(hdwnd, ID_SB1)) {
        // 흘림띠조종체이다.
        cntlpos++;
        if(cntlpos>>VERTRANGEMAX) cntlpos = VERTRANGEMAX;
      }
      else {
        // 표준흘림띠이다.
        vpos++;
        if(vpos>>VERTRANGEMAX) vpos = VERTRANGEMAX;
      }
      break;
    case SB_LINEUP:
      if((HWND)lParam==GetDlgItem(hdwnd, ID_SB1)) {
        // 흘림띠조종체이다.
        cntlpos--;
        if(cntlpos<0) cntlpos = 0;
      }
      else {
        /* 표준흘림띠이다. */
        vpos--;
        if(vpos<0) vpos = 0;
      }
      break;
    case SB_THUMBPOSITION:
      if((HWND)lParam==GetDlgItem(hdwnd, ID_SB1)) {
        // 흘림띠조종체이다.
        cntlpos = HIWORD(wParam); // 현재위치를 얻는다.
      }
      else {
        // 표준흘림띠이다.
        vpos = HIWORD(wParam); // 현재위치를 얻는다.
      }
      break;
```

```
      case SB_THUMBTRACK:
        if((HWND)lParam==GetDlgItem(hdwnd, ID_SB1)) {
          // 흘림띠조종체이다.
          cntlpos = HIWORD(wParam); // 현재위치를 얻는다.
        }
        else {
          // 표준흘림띠이다.
          vpos = HIWORD(wParam); // 현재위치를 얻는다.
        }
        break;
      case SB_PAGEDOWN:
        if((HWND)lParam==GetDlgItem(hdwnd, ID_SB1)) {
          // 흘림띠조종체이다.
          cntlpos += 5;
          if(cntlpos>>VERTRANGEMAX) cntlpos = VERTRANGEMAX;
        }
        else {
          // 표준흘림띠이다.
          vpos += 5;
          if(vpos>>VERTRANGEMAX) vpos = VERTRANGEMAX;
        }
        break;
      case SB_PAGEUP:
        if((HWND)lParam==GetDlgItem(hdwnd, ID_SB1)) {
          // 흘림띠조종체이다.
          cntlpos -= 5;
          if(cntlpos<0) cntlpos = 0;
        }
        else {
          // 표준흘림띠이다.
          vpos -= 5;
          if(vpos<0) vpos = 0;
        }
        break;
    }

    if((HWND)lParam==GetDlgItem(hdwnd, ID_SB1)) {
      // 흘림띠조종체의 위치를 갱신한다.
```

```
      si.fMask = SIF_POS;
      si.nPos = cntlpos;
      SetScrollInfo((HWND)lParam, SB_CTL, &si, 1);
      hdc = GetDC(hdwnd);
      sprintf(str, "Scroll Bar Control: %d    ", cntlpos);
      TextOut(hdc, 1, 60, str, strlen(str));
      ReleaseDC(hdwnd, hdc);
    }
    else {
      // 표준흘림띠의 위치를 갱신한다.
      si.fMask = SIF_POS;
      si.nPos = vpos;
      SetScrollInfo(hdwnd, SB_VERT, &si, 1);
      hdc = GetDC(hdwnd);
      sprintf(str, "Vertical: %d    ", vpos);
      TextOut(hdc, 1, 1, str, strlen(str));
      ReleaseDC(hdwnd, hdc);
    }
    return 1;
  case WM_HSCROLL:
    switch(LOWORD(wParam)) {
      /* 여기에 수평흘림띠의 다른 사건을 처리하는
         프로그람코드를 추가할수 있다. */
      case SB_LINERIGHT:
        hpos++;
        if(hpos>>HORZRANGEMAX) hpos = HORZRANGEMAX;
        break;
      case SB_LINELEFT:
        hpos--;
        if(hpos<0) hpos = 0;
    }
    // 수평흘림띠의 위치를 갱신한다.
    si.fMask = SIF_POS;
    si.nPos = hpos;
    SetScrollInfo(hdwnd, SB_HORZ, &si, 1);
    hdc = GetDC(hdwnd);
    sprintf(str, "Horizontal: %d    ", hpos);
    TextOut(hdc, 1, 30, str, strlen(str));
```

```
            ReleaseDC(hdwnd, hdc);
            return 1;
        }

        return 0;
    }
```

프로그람의 실행결과를 그림 6-2 에 보여 주었다. 수평흐림띠조종체를 추가하는것도 시험해 볼수 있다.

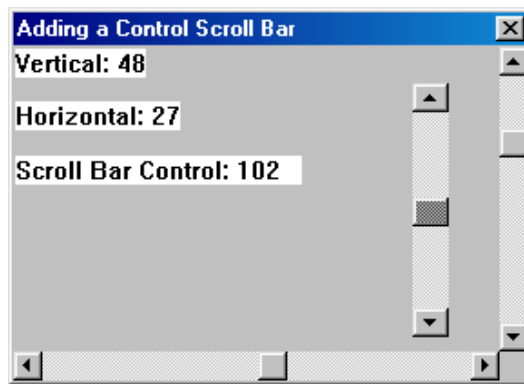



그림 6-2. 흘림띠조종체 실례프로그람의 실행결과

검 사 칸

*검사칸*은 추가선택항목의 선택상태를 설정 혹은 비설정으로 하는데 사용되는 조종체이다. 검사칸은 작은 4 각형모양이며 그 안에 검사표식을 표시하거나 표시하지 않을수 있다. 검사칸은 추가선택항목에 대해 설명하는 표식(Label)을 가지고 있다. 검사칸안에 검사표식이 있는 상태를 검사된 상태(설정)라고 하며 해당 항목이 선택되였다는것을 의미한다. 검사칸이 비여 있는 경우는 해당 항목이 선택되여 있지 않다는것(비설정)을 의미한다.

검사칸은 대화칸에서 사용되는 조종체이며 프로그람의 자원파일에서 대화칸의 정의안에 추가된다. 대화칸의 정의에 검사표식을 추가하려면 CHECKBOX 또는 AUTOCHECKBOX 중 한개의 명령문을 사용한다. 아래에 일반적인 서식을 보여 주었다.

CHECKBOX “string”, CBID, X, Y, Width, Height[, Style]
AUTOCHECKBOX “string”, CBID, X, Y, Width, Height[, Style]

string 은 검사칸과 함께 표시되는 문자렬이고 CBID 는 검사칸을 식별하기 위한 값이다. 검사칸의 왼쪽웃모서리의 자리표를 X, Y 에 설정하고 검사칸과 문자렬을 합친 크기를 Width 및 Height 에 설정한다. Style 에는 검사칸의 형식을 설정한다. 형식을 설정하지 않은 경우는 체계설정으로 검사칸의 오른쪽에 문자렬이 표시되며 검사칸이 건반입력초점을 받게 된다. 검사칸이 작성된 직후는 검사되여 있지 않는 상태로 된다.

Windows 2000 을 다루어 보면 알수 있는것처럼 검사칸의 상태는 *반전절환*되면서 변한다. 검사칸을 선택할 때마다 검사된 상태와 검사되지 않은 상태가 서로 바뀐다.

이 조작은 필요에 따라 자동화할수 있다. *CHECKBOX 명령문*을 사용하면 수동적인 검사칸을 작성할수 있으며 프로그람코드로 검사칸의 검사상태를 설정하게 된다.(다시말하여 수동 검사칸은 프로그람으로 조작하여야 한다.)

*AUTOCHECKBOX 명령문*을 사용하면 자동적인 검사칸을 작성할수 있으며 Windows 2000 이 검사칸의 조작을 맡아 하도록 할수 있다. 자동 검사칸은 그것이 선택될 때마다 Windows 가 자동적으로 상태를 반전절환(검사상태의 설정/비설정)한다. 대다수의 응용프로그람에서는 검사칸을 수동으로 조작할 필요가 없으므로 앞으로 작성하게 될 실례프로그람에서는 AUTOCHECKBOX 만을 사용하기로 한다.

검사칸의 상태를 얻기

검사칸의 검사상태는 설정 혹은 비설정으로 되여 있다. *SendDlgItemMessage( )*라는 API 함수를 리용하여 검사칸에 *BM_GETCHECK 통보문*을 보내면 검사칸의 상태를 얻을수 있다. 5 장에서도 SendDlgItemMessage( )에 대해 설명하였지만 아래에 다시 한번 선언을 보여 주었다.

```
LONG SendDlgItemMessage(HWND hdwnd, int ID, UINT IDMsg,
                          WPARAM wParam, LPARAM lParam);
```

ID에 대화칸안의 조종체를 식별하기 위한 값을 설정하고 IDMsg에 통보문을 설정한다. hdwnd 에 대화칸의 손잡이를 설정한다. *BM_GETCHECK* 를 보내는 경우에는 wParam 과 lParam 의 값을 령으로 한다. 검사칸이 검사되여 있다면 *BST_CHECKED* 가 돌려 지며 그렇지 않은 경우에는 *BST_UNCHECKED* 가 돌려 진다.

검사칸에 검사표식을 붙이기

프로그람에서 검사칸에 검사표식을 붙일수 있다. 그러자면 API 함수 SendDlgItemMessage( )를 리용하여 *BM_SETCHECK 통보문*을 검사칸에 보낸다. 이 경우에는 wParam에 검사칸의 상태를 설정으로 하겠는가 비설정으로 하겠는가를 설정한다.

wParam 이 *BST_CHECKED* 인 경우는 검사칸의 상태가 설정으로 된다. BST_UNCHECKED 인 경우는 검사칸의 상태가 비설정으로 된다. 어떤 경우에나 lParam 에는 령을 설정한다.

이미 설명한것처럼 수동검사칸에서는 사용자의 조작에 응답하여 프로그람코드에서 검사상태를 변경하여야 한다. 그러나 자동검사칸을 사용하면 프로그람에서는 초기상태를 설정해 주기만 하면 된다. 자동검사칸을 사용하는 경우에 그것이 선택될 때마다 상태가 자동적으로 변화한다.

작성직후의 검사칸의 상태는 비설정(검사표식이 붙어 있지 않는)으로 되여 있다. 그러므로 대화칸이 작성되였을 때의 검사칸의 상태는 비설정으로 된다.

검사칸의 직전의 상태를 반영시키려면 대화칸이 작성될 때마다 초기화를 진행하여야 한다. 그렇게 하기 위한 가장 간단한 방법은 대화칸이 WM_INITDIALOG 통보문을 받았을 때 적절한 BM_SETCHECK 통보문을 검사칸에 보내는것이다.

검사칸의 통보문

사용자가 검사칸을 찰칵하거나 검사칸을 선택한 다음 공백건을 누르면 WM_COMMAND 통보문이 대화함수에 보내여 진다. 이때 wParam 의 아래단어에는 검사칸을 식별하는 값이 보관되며 wParam 의 웃단어에는 *BN_CLICKED* 라는 통지문이 보관된다. 수동검사칸을 리용하는 경우에는 이 통보문에 대한 응답으로서 검사칸의 상태를 변경하는 처리를 해야 한다.

다시 한보 전진

3 상태검사칸

Windows 2000 은 3 상태검사칸이라고 하는 검사칸도 제공한다. 이 검사칸은 설정, 비설정 및 무효의 세개의 상태를 가지고 있다.(조종체가 회색으로 표시되여 있는 경우는 무효이다.)

일반적인 검사칸과 같이 3 상태검사칸은 AUTO3STATE 또는 STATE3 라는 자원파일 명령문을 리용하여 자동 혹은 수동조종체로 할수 있다. 아래에 일반적인 서식을 보여 주었다.

```
STATE3 “string”, ID, X, Y, Width, Height[, Style]
AUTO3STATE “string”, ID, X, Y, Width, Height[, Style]
```

string 은 검사칸과 함께 표시되는 문자렬이고 ID 는 검사칸을 식별하는 값이다. 검사칸의 왼쪽웃모서리의 자리표를 X, Y 에 설정하고 검사칸에 문자렬을 합친 크기를 Width 와 Height 에 설정한다. Style 에는 검사칸의 형식을 설정한다. 형식을 설정하지 않으면 체계설정으로 문자렬을 오른쪽에 표시하며 건반입력

초점을 가지게 된다. 3상태검사칸이 작성된 직후의 상태는 비설정으로 되여 있다.

BM_GETCHECK 통보문에 대한 응답으로서 3상태검사칸은 비설정상태라면 BST_UNCHECKED, 설정상태라면 BST_CHECKED, 무효상태라면 BST_INDETERMINATE를 돌려 준다.

이와 마찬가지로 BM_SETCHECK 를 사용하여 3 상태검사칸의 상태를 설정하는 경우에는 BST_UNCHECKED일 때 비설정상태, BST_CHECKED일 때 설정상태, BST_INDETERMINATE일 때 무효상태로 된다.

단일선택단추

다음에 설명할 조종체는 *단일선택단추*이다. 단일선택단추는 서로 상반되는 추가선택항목들을 선택하는데 쓰인다. 단일선택단추는 표식과 작은 동그란 단추로 구성되여 있다. 단추가 Off 로 설정된 경우는 선택되여 있지 않다는것을 가리키며 단추가 On 으로 설정되여 있는 경우는 선택되여 있다는것을 가리킨다.

Windows 2000 은 수동형과 자동형의 두가지 형식의 단일선택단추를 제공한다. 수동단일선택단추에서는 수동검사칸과 마찬가지로 프로그람에서 모든 조작을 진행해야 한다. 자동단일선택단추에서는 이 조작이 자동화되여 있다. 자동단일선택단추의 리용이 압도적이므로 여기에서는 자동단일선택단추에 대하여서만 설명하기로 한다.

다른 조종체와 같이 단일선택단추는 프로그람의 자원파일의 대화칸정의안에서 정의된다. 자동 단일선택단추를 작성하기 위하여 *AUTORADIOBUTTON 명령문*을 리용한다. 아래에 일반적인 구문을 보여 주었다.

AUTORADIOBUTTON “string”, RBID, X, Y, Width, Height[, Style]

string 은 단추와 함께 표시되는 문자렬이고 RBID 는 단일선택단추를 식별하는 값이다. 단일선택단추의 왼쪽웃모서리의 자리표를 X, Y 에 설정하고 단추와 문자렬을 합한 크기를 Width 와 Height 에 설정한다. Style 에는 단일선택단추의 형식을 설정한다. 형식이 설정되지 않은 경우에는 체계설정으로 단추의 오른쪽에 문자렬을 표시하고 건반입력초점을 가지게 된다. 단추는 체계설정으로 설정상태로 되여 있다.

이미 설명한것처럼 단일선택단추는 서로 상반되는 추가선택항목들의 모임을 만드는데 사용된다. 이러한 모임을 만드는 경우에 자동단일선택단추를 사용하면 Windows 가 한 단추만이 설정되도록 자동적으로 관리해 준다. 다시말하여 사용자가 한 단추를 선택하면 바로 전에 선택되여 있던 다른 단추의 상태가 비설정으로 된다. 사용자가 동시에

여러개의 단추를 선택할수는 없다.

단일선택단추(자동단일선택단추도 포함)의 상태는 프로그람에서 SendDlgItem Message( )를 리용하여 *BM_SETCHECK 통보문*을 보내서 설정할수 있다. 이때 wParam에는 단추의 설정 혹은 비설정상태를 설정한다. wParam에 *BST_CHECKED*를 설정하면 단추의 상태가 설정으로 된다. *BST_UNCHECKED* 를 설정하면 단추의 상태는 비설정으로 된다. 체계설정으로는 모든 단추가 비설정으로 되여 있다.

*BM_GETCHECK 통보문*을 보내여 단일선택단추의 상태를 얻을수 있다. 단추가 설정된 상태이라면 *BST_CHECKED* 가 돌려 지고 비설정상태이라면 *BST_UNCHECKED* 가 돌려 진다.

사용자가 단일선택단추를 조작하면 WM_COMMAND 통보문이 발송된다. 수동단일선택단추를 사용하지 않는다면 보통 이 통보문에 응답할 필요는 없다.

참고 *: SendDlgItemMessage( )를 리용하면 한개이상의 단추를 설정상태로 하거나 모두 비설정상태로 만들수도 있다. 그러나 일반적인 Windows 의 류의에서는 단일선택단추가 서로 상반되는 항목을 선택하는데 사용되므로 하나의 항목만이 선택되도록 해야 한다. 이 규칙을 지킬것을 강하게 권고한다.*

검사칸, 단일선택단추 및 흘림띠의 실례프로그람

이 절에서는 *검사칸*, *단일선택단추* 및 *흘림띠*의 실례를 보여 주기 위해 간단하면서도 편리한 프로그람을 작성해 보기로 한다. 이 프로그람은 내려세기시계를 실현하는것이다.

시계를 사용하기 위해 시간을 s 단위로 설정한다. 프로그람은 설정시간의 경과를 감시하다가 시간이 되였다는것을 알려 준다. 이 내려세기시계에서는 시간의 설정을 흘림띠로 진행한다. 검사칸과 단일선택단추는 시간이 경과했을 때 경보음을 울리겠는가 울리지 않겠는가 등 여러가지 추가선택항목들을 설정하는데 리용된다.

프로그람을 작성하기에 앞서 Windows 2000 의 기능의 하나인 시계에 대해 배워 두자. 내려세기시계프로그람에서는 시간의 경과를 검사하기 위하여 Windows 에 미리 갖추어 진 시계를 한개 사용한다.

시계통보문의 생성

Windows 2000 에서는 일정한 시간간격으로 프로그람에 새치기를 거는 *시계*를 작성할수 있다. 지정된 시간간격으로 Windows 2000 은 프로그람에 WM_TIMER 통보문을 보낸다. 시계는 특히 과제를 배경에서 실행시킬 때 편리하다.

시계를 기동하자면 *SetTimer( )*라는 API 함수를 사용한다. 선언은 다음과 같다.

```
UINT SetTimer(HWND hwnd, UINT nID, UINT wLength,
              TIMERPROC lpTFunc);
```

hwnd는 시계를 사용하는 창문의 손잡이이다. 일반적으로 이 창문은 프로그람의 기본창문이거나 대화칸의 창문이다. nID에는 시계를 식별하는 값을 설정한다.(여러개의 시계를 사용하는 경우에는 식별이 필요하다.) wLength에는 ms단위로 시간간격을 설정한다. 다시말하여 wLength가 새치기하는 시간간격을 결정한다.

lpTFunc에 설정되는 함수의 지시자는 시계의 시간이 경과했을 때 호출되는 시계함수이다. 그러나 lpTFunc에 NULL을 설정하면 hwnd에 설정된 창문의 창문함수가 호출되게 되므로 자체의 시계함수를 준비할 필요는 없다. 이 경우에는 시계의 시간이 경과하면 프로그람의 통보문대기렬에 WM_TIMER가 보관되며 다른 통보문과 꼭같이 처리할 수 있다.

뒤에서 보게 될 실례프로그람에서는 이 수법이 사용된다. SetTimer( )함수의 호출이 성공하면 nID의 값이 돌려 진다. 시계가 할당되지 않은 경우는 령이 돌려 진다.

독자의 시계함수를 작성하려는 경우에는 그것을 아래에서 보여 주는 선언과 같이 *역호출함수*로서 작성하여야 한다.(물론 함수의 이름은 임의로 하여도 상관 없다.)

```
VOID CALLBACK Tfunc(HWND hwnd, UINT msg, UINT TimerID,
                    DWORD SysTime);
```

hwnd에는 시계를 사용하는 창문의 손잡이가, msg에는 WM_TIMER 통보문, TimerID에는 시계를 식별하는 값, sysTime에는 현재 체계시간이 보관된다.

시계가 일단 기동되면 응용프로그람을 끝내든가 프로그람이 *KillTimer( )*라는 API 함수를 호출할 때까지 시계의 새치기가 계속된다. *KillTimer( )*의 선언은 다음과 같다.

```
BOOL KillTimer(HWND hwnd, UINT nID);
```

hwnd는 시계를 사용하는 창문의 손잡이이며 nID는 시계를 식별하는 값이다. 함수의 호출이 성공하면 령 아닌 값을 돌려 지며 실패하면 령이 돌려 진다.

WM_TIMER 통보문이 생성되였을 때 wParam에는 시계의 ID가 보관되고 lParam에는 만일 존재한다면 시계의 역호출함수의 주소가 보관된다. 내려세기시계프로그람에서 lParam은 NULL로 된다.

내려세기시계프로그람의 자원파일과 머리부파일

내려세기시계프로그람은 아래와 같은 자원파일을 사용한다.

```
// 흘림띠,검사칸 및 단일선택단추의 실례
#include <windows.h>
#include "cd.h"

MyMenu MENU
{
  POPUP "&Dialog"
  {
    MENUITEM "&Timer\tF2", IDM_DIALOG
    MENUITEM "&Exit\tF3", IDM_EXIT
  }
  MENUITEM "&Help", IDM_HELP
}

MyMenu ACCELERATORS
{
  VK_F2, IDM_DIALOG, VIRTKEY
  VK_F3, IDM_EXIT, VIRTKEY
  VK_F1, IDM_HELP, VIRTKEY
}

MyDB DIALOGEX 18, 18, 152, 92
CAPTION "A Countdown Timer"
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
      | WS_VSCROLL
{
  PUSHBUTTON "Start", IDD_START, 10, 60, 30, 14
  PUSHBUTTON "Cancel", IDCANCEL, 60, 60, 30, 14
  AUTOCHECKBOX "Show Countdown", IDD_CB1, 1, 20, 70, 10
  AUTOCHECKBOX "Beep At End", IDD_CB2, 1, 30, 60, 10
  AUTORADIOBUTTON "Minimize", IDD_RB1, 80, 20, 50, 10
  AUTORADIOBUTTON "Maximize", IDD_RB2, 80, 30, 50, 10
  AUTORADIOBUTTON "As-Is", IDD_RB3, 80, 40, 50, 10
}
```

내려세기시계프로그람에서 사용되는 머리부파일을 아래에 보여 주었다. 이 파일을 CD.H라는 파일이름으로 작성해 둔다.

```
#define IDM_DIALOG          100
#define IDM_EXIT            101
```

```
#define IDM_HELP             102

#define IDD_START            300
#define IDD_TIMER            301

#define IDD_CB1              400
#define IDD_CB2              401
#define IDD_RB1              402
#define IDD_RB2              403
#define IDD_RB3              404
```

내려세기시계의 프로그람코드

내려세기시계프로그람의 완전한 프로그람코드를 실례 6-3 에 주었다.

실례 6-3. Cd 프로그람

```
// 내려세기시계

#include <windows.h>
#include <cstring>
#include <cstdio>
#include "cd.h"

#define VERTRANGEMAX 200

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK DialogFunc(HWND, UINT, WPARAM, LPARAM);

char szWinName[] = "MyWin"; // 창문클라스의 이름

HINSTANCE hInst;

HWND hwnd;

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  MSG msg;
  WNDCLASSEX wcl;
  HACCEL hAccel;

  // 창문클라스를 정의한다.
```

```
wcl.cbSize = sizeof(WNDCLASSEX);

wcl.hInstance = hThisInst;      // 실체의 손잡이
wcl.lpszClassName = szWinName; // 창문클라스의 이름
wcl.lpfnWndProc = WindowFunc;  // 창문함수
wcl.style = 0;                 // 체계설정형식

wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

// 차림표자원의 이름을 지정한다.
wcl.lpszMenuName = "MyMenu";

wcl.cbClsExtra = 0;        // 보조기억기령역은 불필요함
wcl.cbWndExtra = 0;

// 창문의 배경색은 흰색으로 한다.
wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

// 창문클라스를 등록한다.
if(!RegisterClassEx(&wcl)) return 0;


/* 창문클라스가 등록되였으므로
    창문을 작성할수 있다. */
hwnd = CreateWindow(
  szWinName, // 창문클라스의 이름
  "Demonstrating Controls", // 제목
  WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
  CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 창문의 너비는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 높이는 Windows가 결정하게 한다.
  NULL,          // 어미창문은 없다.
  NULL,          // 클라스차림표의 덧쓰기는 하지 않는다.
  hThisInst,     // 실체의 손잡이
  NULL           // 추가파라메터는 없다.
);

hInst = hThisInst; // 현재 실체의 손잡이를 보관한다.

// 가속기를 적재한다.
```

```
    hAccel = LoadAccelerators(hThisInst, "MyMenu");

    // 창문을 표시한다.
    ShowWindow(hwnd, nWinMode);
    UpdateWindow(hwnd);

    // 통보문순환고리를 작성한다.
    while(GetMessage(&msg, NULL, 0, 0))
    {
      if(!TranslateAccelerator(hwnd, hAccel, &msg)) {
        TranslateMessage(&msg); // 건반통보를 변환한다.
        DispatchMessage(&msg);  // Windows 2000에 조종을 넘긴다.
      }
    }

    return msg.wParam;
  }

  /* 이 함수는 Windows 2000으로부터 호출되여
     통보문대기렬에서 꺼낸 통보문을 받는다.
  */
  LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                              WPARAM wParam, LPARAM lParam)
  {
    int response;

    switch(message) {
      case WM_COMMAND:
        switch(LOWORD(wParam)) {
          case IDM_DIALOG:
            DialogBox(hInst, "MyDB", hwnd, (DLGPROC) DialogFunc);
            break;
          case IDM_EXIT:
            response = MessageBox(hwnd, "Quit the Program?",
                                  "Exit", MB_YESNO);
            if(response == IDYES) PostQuitMessage(0);
            break;
          case IDM_HELP:
            MessageBox(hwnd, "Try the Timer", "Help", MB_OK);
            break;
```

```
        }
        break;
      case WM_DESTROY: // 프로그람을 끝낸다.
        PostQuitMessage(0);
        break;
      default:
        /* 이 switch 문에서 지정된것 이외의 통보문은
           Windows 2000 이 처리하게 한다. */
        return DefWindowProc(hwnd, message, wParam, lParam);
    }
    return 0;
}

// 대화함수
BOOL CALLBACK DialogFunc(HWND hdwnd, UINT message,
                         WPARAM wParam, LPARAM lParam)
{
    char str[80];
    static int vpos = 0;  // 수직흘림칸의 위치
    static SCROLLINFO si; // 흘림띠정보를 보관하는 구조체

    HDC hdc;
    PAINTSTRUCT paintstruct;

    static int t;

    switch(message) {
      case WM_COMMAND:
        switch(LOWORD(wParam)) {
          case IDCANCEL:
            EndDialog(hdwnd, 0);
            return 1;
          case IDD_START: // 시계를 기동한다.
            SetTimer(hdwnd, IDD_TIMER, 1000, NULL);
            t = vpos;
            if(SendDlgItemMessage(hdwnd,
                   IDD_RB1, BM_GETCHECK, 0, 0) == BST_CHECKED)
              ShowWindow(hwnd, SW_MINIMIZE);
            else
            if(SendDlgItemMessage(hdwnd,
```

```
                IDD_RB2, BM_GETCHECK, 0, 0) == BST_CHECKED)
              ShowWindow(hwnd, SW_MAXIMIZE);
            return 1;
        }
        break;
    case WM_TIMER: // 시계의 시간이 경과했다.
        if(t==0) {
          KillTimer(hdwnd, IDD_TIMER);
          if(SendDlgItemMessage(hdwnd,
                  IDD_CB2, BM_GETCHECK, 0, 0) == BST_CHECKED)
            MessageBeep(MB_OK);
          else MessageBox(hdwnd, "Timer Went Off", "Timer", MB_OK);
          ShowWindow(hwnd, SW_RESTORE);
          return 1;
        }
        t--;

        // 내려세기를 표시할것인가를 확인한다.
        if(SendDlgItemMessage(hdwnd,
                IDD_CB1, BM_GETCHECK, 0, 0) == BST_CHECKED) {
          hdc = GetDC(hdwnd);
          sprintf(str, "Counting: %d   ", t);
          TextOut(hdc, 1, 1, str, strlen(str));
          ReleaseDC(hdwnd, hdc);
        }
        return 1;
    case WM_INITDIALOG:
        si.cbSize = sizeof(SCROLLINFO);
        si.fMask = SIF_RANGE;
        si.nMin = 0; si.nMax = VERTRANGEMAX;
        SetScrollInfo(hdwnd, SB_VERT, &si, 1);

        // 「As-Is」단일선택단추를 확인한다.
        SendDlgItemMessage(hdwnd, IDD_RB3, BM_SETCHECK, BST_CHECKED, 0);
        return 1;
    case WM_PAINT:
        hdc = BeginPaint(hdwnd, &paintstruct);
        sprintf(str, "Interval: %d", vpos);
        TextOut(hdc, 1, 1, str, strlen(str));
```

```
        EndPaint(hdwnd, &paintstruct);

        return 1;

      case WM_VSCROLL:

        switch(LOWORD(wParam)) {

          case SB_LINEDOWN:

            vpos++;

            if(vpos>>VERTRANGEMAX) vpos = VERTRANGEMAX;

            break;

          case SB_LINEUP:

            vpos--;

            if(vpos<0) vpos = 0;

            break;

          case SB_THUMBPOSITION:

            vpos = HIWORD(wParam); // 현재위치를 얻는다.

            break;

          case SB_THUMBTRACK:

            vpos = HIWORD(wParam); // 현재위치를 얻는다.

            break;

          case SB_PAGEDOWN:

            vpos += 5;

            if(vpos>>VERTRANGEMAX) vpos = VERTRANGEMAX;

            break;

          case SB_PAGEUP:

            vpos -= 5;
```

```
            if(vpos<0) vpos = 0;
        }
        si.fMask = SIF_POS;
        si.nPos = vpos;
        SetScrollInfo(hdwnd, SB_VERT, &si, 1);
        hdc = GetDC(hdwnd);
        sprintf(str, "Interval: %d    ", vpos);
        TextOut(hdc, 1, 1, str, strlen(str));
        ReleaseDC(hdwnd, hdc);
        return 1;
    }
    return 0;
}
```

프로그람의 실행결과를 그림 6-3 에 보여 주었다.

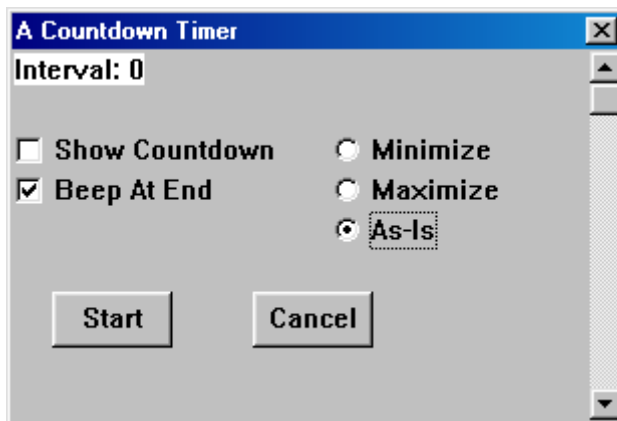



그림 6-3. 내려세기시계프로그람의 실행결과

내려세기시계프로그람의 상세

내려세기시계프로그람에서 사용되는 조종체들이 어떻게 조작되는가를 보자.

수직흘림띠는 시간간격을 설정하는데 사용되고 있다. 흘림띠의 상태를 검사하는 부분에서는 앞에서 설명한것과 같은 프로그람코드가 사용되므로 더 설명할 필요가 없다. 그러나 검사칸과 단일선택단추를 조작하는 프로그람코드는 자세히 볼 필요가 있다.

이미 설명한바와 같이 작성된 직후에는 어느 단일선택단추도 설정되여 있지 않다. 그러므로 대화칸이 작성되였을 때 프로그람에서 어느 하나를 설정해 주어야 한다. 이 실례프로그람에서는 WM_INITDIALOG 통보문이 전송될 때마다 아래의 프로그람코드를 사용하여 [As-Is]단일선택단추(IDD_RB3)를 설정한다.

```
SendDlgItemMessage(hdwnd, IDD_RB3, BM_SETCHECK,
                                    BST_CHECKED, 0);
```

시계를 기동하기 위하여 사용자는 [Start]단추를 누른다. 이것은 아래의 프로그람코드로 실행된다.

```
        case IDD_START: // 시계를 기동한다.
          SetTimer(hdwnd, IDD_TIMER, 1000, NULL);
          t = vpos;
          if(SendDlgItemMessage(hdwnd,
                  IDD_RB1, BM_GETCHECK, 0, 0) == BST_CHECKED)
             ShowWindow(hwnd, SW_MINIMIZE);
          else
          if(SendDlgItemMessage(hdwnd,
                  IDD_RB2, BM_GETCHECK, 0, 0) == BST_CHECKED)
             ShowWindow(hwnd, SW_MAXIMIZE);
      return 1;
```

시계의 시간은 1s(1000 ㎳)로 설정되였다. 계수기로 되는 변수 t 에는 수직흘림띠의 위치값이 설정된다. [Minimize]단일선택단추가 설정된 경우는 프로그람창문이 최소화된다. [Maximize]단일선택단추가 설정된 경우는 프로그람창문이 최대화된다. 그밖의 경우에 프로그람의 창문크기는 변하지 않는다.

대화칸의 손잡이 hdwnd 가 아니라 기본창문의 손잡이 hwnd 를 지정하여 ShowWindow( )를 호출한다는점에 주목해야 한다. 프로그람을 최소화 또는 최대화하는데는 대화칸의 손잡이가 아니라 기본창문의 손잡이가 사용된다.

또한 이 프로그람에서 hdwnd 가 대역변수로 선언되여 있는 점에도 주목해야 한다. 이렇게 하여 DialogFunc( )에서 hdwnd 을 사용할수 있다.

WM_TIMER 통보문을 받을 때마다 아래의 프로그람코드가 실행된다.

```
case WM_TIMER: // 시간이 경과했다.
        if(t==0) {
          KillTimer(hdwnd, IDD_TIMER);
          if(SendDlgItemMessage(hdwnd,
                   IDD_CB2, BM_GETCHECK, 0, 0) == BST_CHECKED)
            MessageBeep(MB_OK);
          else MessageBox(hdwnd, "Timer Went Off", "Timer", MB_OK);
          ShowWindow(hwnd, SW_RESTORE);
          return 1;
        }
        t--;

        // 내려세기를 표시할것인가를 검사한다.
        if(SendDlgItemMessage(hdwnd,
                 IDD_CB1, BM_GETCHECK, 0, 0) == BST_CHECKED) {
          hdc = GetDC(hdwnd);
          sprintf(str, "Counting: %d   ", t);
          TextOut(hdc, 1, 1, str, strlen(str));
          ReleaseDC(hdwnd, hdc);
          }
      return 1;
```

내려세기가 령에 도달하면 시계가 정지되고 필요에 따라 창문이 본래의 크기로 돌아간다. [Beep At End]단추가 설정되면 *MessageBeep( )*라는 API 함수를 사용하여 콤퓨터의 고성기가 경보음을 내게 된다. 그렇지 않은 경우에는 통보칸이 표시된다. 설정시간에 도달하지 않은 경우에는 계수기인 변수 t 가 감소된다. [Show Countdown]단추가 설정되면 남은 시간이 표시된다.

MessageBeep( )의 선언은 다음과 같다.

BOOL MessageBeep(UINT sound);

sound 에는 소리의 종류를 설정한다. 이 값을 -1 로 하면 표준적인 경보음이 울리며 아래의 값을 설정할수도 있다.

MB_ICONASTERISK	MBICONEXCLAMATION	MB_OK
MB_ICONHAND	MB_ICONQUESTION	

MB_OK 도 표준적인 경보음이 울리게 한다. 호출이 성공하면 MessageBeep( )는 령 아닌 값을 돌려 주며 실패하면 령을 돌려 준다.

내려세기시계프로그람을 통해 알수 있는바와 같이 자동검사칸과 단일선택단추는 대체로 Window 2000 에 의해 조작되므로 그 조종체들을 처리하기 위한 프로그람코드는 놀

라울 정도로 작다. 사용방법이 간단하다는것으로 하여 검사칸이나 단일선택단추는 많은 상황에서 사용된다.

정적조종체

표준적인 조종체들가운데서 가장 간단하게 사용되는것은 정적조종체이다. *정적조종체*는 통보문을 생성하지도 않으며 받지도 않는다. 다시말하여 정적조종체의 역할이란 대화칸에 어떤 요소를 표시하는것뿐이다.

정적조종체에는 정적본문으로 되는 *CTEXT*, *RTEXT*, *LTEXT* 및 조종체를 집합으로 묶는데 사용되는 *GROUPBOX*가 있다.

CTEXT 조종체는 지정된 령역의 중심에 맞추어 문자렬을 표시한다. *LTEXT*는 문자렬을 왼쪽맞추기하여 표시한다. *RTEXT* 는 문자렬을 오른쪽맞추기하여 표시한다. 이 조종체들의 일반적인 서식을 아래에 보여 주었다.

```
CTEXT "text", ID, X, Y, Width, Height[, Style]
RTEXT "text", ID, X, Y, Width, Height[, Style]
LTEXT "text", ID, X, Y, Width, Height[, Style]
```

text 에는 표시할 문자렬을 설정한다. ID 에는 문자렬을 식별하기 위한 값을 설정한다. 표시되는 문자렬의 왼쪽웃모서리의 자리표를 X, Y 에 설정하고 크기를 Width 와 Height 에 설정한다. Style 에는 조종체의 형식을 설정한다. 일반적으로는 체계설정의 형식을 사용한다.

문자렬을 둘러 싸는 4 각형은 표시되지 않는다. X, Y, Width 및 Height 는 문자렬이 점유하는 령역을 지정하기 위한 목적으로만 리용된다.

정적본문은 대화칸에 문자렬을 표시할 때 편리하게 쓰인다. 정적본문은 대화칸안의 다른 조종체의 표식으로서 사용되거나 사용자에게 간단한 설명을 표시하는데 잘 사용된다.

집합칸은 대화칸안의 조종체들을 둘러 싸고 그것들을 집합하는데 사용된다. 또한 집합칸은 제목을 표시할수 있다. *GROUPBOX*의 일반적인 구문은 아래와 같이 되여 있다.

```
GROUPBOX "title", ID, X, Y, Width, Height[, Style]
```

title 에는 집합칸의 제목을 설정한다. ID 에는 집합칸을 식별하는 값을 설정한다. 집합칸의 왼쪽웃모서리의 자리표를 X, Y 에 설정하고 크기를 Width 와 Height 에 설정한다. Style 에는 집합칸의 형식을 설정한다.

일반적으로는 체계설정의 형식이 사용된다.

집합칸의 기능을 확인하기 위하여 내려세기시계프로그람의 자원파일에 아래의 정의

를 추가한다.

GROUPBOX "Display As", 1, 72, 10, 60, 46

집합칸을 추가하면 대화칸은 그림 6-4 와 같이 된다. 집합칸을 사용하여 대화칸의 외형을 변경시켜도 프로그람의 기능은 변하지 않는다.

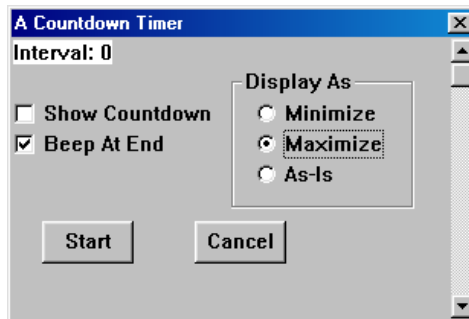



그림 6-4. 집합칸을 추가한 내려세기프로그람

다시 한보 전진

단독의 조종체

조종체는 대화칸안에서 사용되는것이 일반적이지만 기본창문안에서 단독으로 리용할수도 있다. 이러한 독자적인 조종체를 작성하려면 CreateWindow( )를 사용하여 목적하는 조종체의 클라스이름과 형식을 지정한다.

아래에 표준적인 조종체들의 클라스이름을 보여 주었다.

BUTTON
COMBOBOX
EDIT
LISTBOX
SCROLLBAR
STATIC

매개 클라스에는 조종체의 형식을 설정하기 위한 마크로가 있다. 이 마크로들에 대한 자세한 설명을 얻으려면 WINDOW.H 또는 API 참고서를 보면 된다.

아래의 프로그람코드는 독자적인 흘림띠와 누름단추를 작성하는것이다.

```
hsbwnd = CreateWindow(
    "SCROLLBAR",    // 흘림띠의 클라스이름
    "",             // 제목이 없다.
    SBS_HORZ | WS_CHILD | WS_VISIBLE,     // 수평흘림띠
    10, 10,                        // 위치
```

```
        120, 20,                    // 크기
        hwnd,                       // 어미창문
        NULL,                       // 흘림띠에는 조종체를 식별하는 값이 필요 없다.
        hThisInst,                  // 프로그람의 실체손잡이
        NULL                        // 추가하는 파라메터는 없다.
    );

    hpbwnd = CreateWindow(
        "BUTTON",                   // 누름단추의 클라스이름
        "Push Button",              // 단추의 내부에 표시되는 문자렬
        BS_PUSHBUTTON | WS_CHIlD | WS_VISIBLE, // 누름단추
        10, 60,                             // 위치
        90, 30,                             // 크기
        hwnd,                               // 어미창문
        (HMENU) 500,                        // 조종체를 식별하는 값
        hThisInst,                          // 프로그람의 실체손잡이
        NULL                                // 추가하는 파라메터는 없다.
    );
```

누름단추의 작성에서는 그것을 식별하는 값이 CreateWindows( )의 9개의 파라메터의 하나로서 설정되여 있다. 이 파라메터에 클라스차림표를 덧쓰기할때 차림표의 손잡이를 설정하였다는것을 상기해 볼 필요가 있다. 조종체를 작성할 때는 이 파라메터에 조종체를 식별하는 값을 설정한다.

이 파라메터는 HMENU형으로 형변환하여야 한다.

독자적인 조종체가 생성한 통보문은 그의 어미창문에 전달된다. 이 통보문을 처리하는 프로그람코드는 어미창문의 창문함수에 서술한다.

제 7 장

비트매프의 사용법과 다시그리기문제의 해결방법

비트매프란 화상을 포함한 객체를 말한다. 이 말은 비트매프가 화상을 정의하는 비트들의 모임이라는데서 유래된것이다. Windows 는 도형에 기초한 조작체계이므로 응용프로그람의 자원으로서 도형을 리용하는것이 보편적이다.

그러나 단지 자원파일에 도형을 포함시키는것만으로 그것을 응용프로그람에서 사용할수 있는것은 아니다. 도형은 Windows 의 사용자대면부를 조작하는 여러가지 기능들의 우에 성립되여 있다.

이 장에서는 Windows 프로그람을 작성할 때 《창문의 다시그리기》를 해결하는 방법에 대해서도 취급한다.

Windows 에는 아이콘과 유표라는 특수한 비트매프도 있다. 아이콘은 어떤 자원 혹은 객체를 가리키며 유표는 마우스의 현재위치를 가리킨다. 지금까지는 미리 갖추어진 아이콘과 유표만을 사용할수 있었다. 이 장에서는 전용아이콘이나 전용유표를 작성하는 방법에 대해서도 설명한다.

비트매프의 두가지 종류

Windows 2000 이 지원하는 *비트매프*에는 *장치의존비트매프*(*DDB*)와 *장치독립비트매프*(*DIB*)의 두가지 종류가 있다. 장치의존비트매프는 특정한 장치용으로 설계된 비트매프이다. 장치독립비트매프는 특정한 장치에 의존하지 않는다.

초기의 Windows에서는 장치의존비트매프만을 취급할수 있었다. Windows 3.0 이후의 모든 판본의 Windows에서는 장치독립비트매프도 사용할수 있게 되여 있다.

장치독립비트매프는 비트매프가 작성된 장치이외의 환경에서 비트매프를 취급하는 경우에 유용하다. 실례로 비트매프를 배포하는 경우에는 그것을 장치독립비트매프로 하는것이 최량의 방법이다. 그러나 특정한 프로그람내에서만 사용되는 비트매프라면 장치의존비트매프라고 해도 충분하다. 이런 리유로 지금까지도 장치의존비트매프가 광범히 사용되고 있다. Win32는 필요에 따라 DDB와 DIB를 변환하는 몇개의 함수를 제공하고 있다.

DDB 와 DIB 의 구조는 서로 다르다. 그러나 이 장에서는 그러한 차이가 중요하지 않다. 사실 응용프로그람의 외형을 설정할 때 비트매프의 2 진형식을 의식하는 일은 거의나 없다. Windows 가 비트매프를 조작하기 위한 고급한 기능의 API 함수를 제공하여 주므로 프로그람작성자는 비트매프의 내부적인 구조를 알 필요는 없다.

이 장에서는 비트매프를 작성한 프로그람에서 그 비트매프를 사용하므로 장치의존비트매프를 사용하기로 한다.

비트매프를 취급하는 두가지 방법

비트매프는 자원에 정의하는 방법과 프로그람내에서 동적으로 작성하는 방법의 두가지 방법으로 얻을수 있다. 첫번째 방법에서 *비트매프자원*이란 프로그람의 외부에서 정의된 화상을 프로그람의 자원파일로 지정한것이다. 두번째 방법의 동적비트매프란 실행시에 프로그람에 의해 작성되는것이다. 이 장에서는 두가지 방법을 다 설명한다. 비트매프자원에 대한 설명으로부터 시작한다.

비트매프자원의 사용방법

비트매프자원을 사용하기 위한 일반적인 순서는 다음과 같다.

- 프로그람의 자원파일에 비트매프를 지정한다.
- 프로그람에서 비트매프를 적재한다.
- 장치상황에 비트매프를 선택한다.

다음 절에서는 이러한 단계를 실현하는 방법을 설명한다.

비트매프자원의 작성

비트매프자원은 앞의 장들에서 설명한 대화칸 및 조종체 등의 자원과는 다르다. 지금까지 설명한 자원들은 자원파일안에 본문 명령문으로 정의하였다. 이와는 달리 비트매프자원은 독립적인 비트매프파일에 보관된 화상으로 되여 있다. 프로그람의 자원파일에는 비트매프파일에 대한 참조를 정의한다.

비트매프자원은 임의의 *화상편집기*를 리용하여 작성된다. 화상편집기는 번역프로그람에 첨부되여 있다. 화상편집기로는 비트매프를 확대하여 표시할수도 있다. 화상편집기를 사용하면 비트매프의 작성이나 편집을 손 쉽게 할수 있다. 실례로 Visual C++의 화상편집기에서 비트매프를 표시한 모양을 그림 7-1 에 보여 주었다.

아이콘이나 유표 등의 특수한 비트매프를 제외하고는 크기를 임의로 할수 있다. 뒤에서 작성하게 되는 실례프로그람에서 비트매프의 크기는 256 × 128 화소로 되여 있다. 이 비트매프파일을 BP.BMP 라는 파일이름으로 작성한다.

이 장에서 작성하는 실례프로그람에서는 그림 7-1 에 보여 준것과 같은 비트매프를 사용한다. 비트매프를 작성하고 다음의 행을 포함하는 BP.RC 라는 자원파일을 작성한다.

```
MyBP BITMAP BP.BMP
```

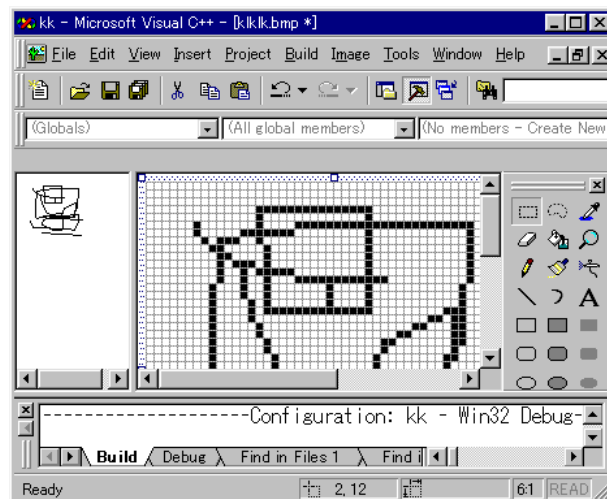



그림 7-1. Visual C++의 화상편집기

우의 행에서 *BITMAP 명령문*은 BP.BMP 파일에 보관된 비트매프자원에 MyBP 라는 이름을 붙여 정의하고 있다. 아래에 *BITMAP 명령문*의 일반적인 서식을 보여 주었다.

```
BitmapName BITMAP Filename
```

BitmapName 은 비트매프를 식별하는 이름이다. 이 이름은 프로그람에서 비트매프를 참조할 때 사용된다. Filename 은 비트매프가 보관된 파일의 이름이다.

비트매프의 표시

비트매프를 작성하고 프로그람의 자원파일에 정의하였다면 그것을 창문의 의뢰자구역에 표시할수 있다. 그를 위한 프로그람코드를 써 넣어야 한다. 여기에서는 비트매프를 표시하는 방법을 설명한다.

비트매프를 표시하려면 먼저 비트매프를 적재하고 그의 손잡이를 얻어야 한다. 이 처리는 WinMain( )내에서 또는 응용프로그람의 기본창문이 *WM_CREATE* 통보문을 받을 때 진행한다. WM_CREATE 통보문은 창문(혹은 그와 관련된 요소)에 대해 어떤 초기화처리를 진행하기 위한 절호의 시점이다.

기본창문의 의뢰자구역에 비트매프자원을 표시하려면 기본창문이 WM_CREATE 통보문을 받을 때 비트매프의 적재를 진행해야 한다. 여기에서는 이 수법을 적용하기로 한다.

비트매프를 적재하는 가장 간단한 방법은 *LoadBitmap( )*라는 API 함수를 사용하는 것이다. 아래에 선언을 보여 주었다.

```
HBITMAP LoadBitmap(HINSTANCE hThisInst,     LPCSTR lpszName);
```

hThisInst 에는 실체의 손잡이를 설정하고 자원파일에 정의되여 있는 비트매프의 이름을 lpszName 에 설정한다. 이 함수는 비트매프의 손잡이를 돌려 주며 오유가 발생한 경우는 NULL 을 돌려 준다. 아래에 실례를 보여 주었다.

```
HBITMAP hbit;        // 비트매프의 손잡이
// ...
hbit = LoadBitmap(hInst, "MyBP");  // 비트매프를 적재한다.
```

이 프로그람코드는 MyBP 라는 이름의 비트매프를 적재하고 그의 손잡이를 hbit 에 보관한다. 비트매프를 표시할 시점으로 되면 프로그람은 아래의 순서로 처리를 진행한다.

- 창문에 출력을 진행하기 위한 장치상황을 얻는다.
- 표시할 비트매프를 보관해 두기 위해 창문의 장치상황과 호환성 있는 *기억기장치상황*을 얻는다.(비트매프는 창문에 복사되기전에 기억기에 보관된다.)

- 기억기장치상황에 비트매프를 선택한다.
- 기억기장치상황으로부터 창문의 장치상황에 비트매프를 복사한다.

이 4 개의 처리를 실행하는 프로그람코드를 다음에 보여 주었다. 여기에서는 WM_PAINT 통보문을 받았을 때 창문의 다른 위치에 두개의 비트매프를 표시하고 있다.

```
HDC hdc, memdc;
PAINTSTRUCT ps;

//...

case WM_PAINT:
     hdc = BeginPaint(hwnd, &ps); // 장치상황을 얻는다.

     memdc = CreateCompatibleDC(hdc); //호환성 있는 장치상황을 작성한다.
     SelectObject(memdc, hbit);  // 비트매프를 선택한다.

     BitBlt(hdc, 10, 10, 256, 128,
                   memdc, 0, 0, SRCCOPY); // 화상을 표시한다.
     BitBlt(hdc, 300, 100, 256, 128,
                   memdc, 0, 0, SRCCOPY); // 화상을 표시한다.
     EndPaint(hwnd, &ps);  // 장치상황을 해제한다.
     DeleteDC(memdc);         // 기억기장치상황을 해제한다.
     break;
```

이 프로그람코드의 내용을 차례로 설명해 보자.

우선 두개의 *장치상황*의 손잡이가 선언되고 있다. hdc 에는 BeginPaint( )에서 얻은 창문의 장치상황이 설정된다. memdc 에는 창문에 그려질 때까지 비트매프를 보관해 두는 기억기장치상황이 설정된다.

WM_PAINT 의 case 문에서 창문의 장치상황이 얻어 진다. 이것이 필요한 리유는 장치상황을 얻지 않으면 창문의 의뢰자구역에 비트매프를 그릴수 없기때문이다.

다음 비트매프를 보관해 두는 *기억기장치상황*이 작성된다. 이 기억기장치상황은 창문의 장치상황과 호환성 있는것이여야 한다. 호환성 있는 기억기장치상황은 *CreateCompatibleDC( )*라는 API 함수를 사용하여 작성된다. 아래에 선언을 보여 주었다.

```
HDC CreateCompatibleDC(HDC hdc);
```

이 함수는 hdc 에 지정된 창문의 장치상황과 호환성 있는 기억기장치상황을 작성하고 그의 손잡이를 돌려 준다. 이 기억기는 표시되기전의 화상을 보관할 때 사용된다. 오유가 발생한 경우에 함수의 돌림값은 NULL 로 된다.

다음 *SelectObject()*라는 API 함수를 사용하여 비트매프를 기억기장치상황에 선택한다. 아래에 선언을 보여 주었다.

```
HGDIOBJ SelectObject(HDC hdc, HGDIOBJ hObject);
```

hdc 에는 장치상황을 설정하고 hObject 에는 장치상황에 선택하는 객체의 손잡이를 설정한다. 이 함수는 (만일 존재한다면) 바로 전에 선택되여 있었던 객체의 손잡이를 돌려 주므로 필요하다면 후에 본래의 상태로 돌아갈수도 있다.

실제로 비트매프를 표시하는데는 *BitBlt( )*라는 API 함수를 사용한다. 이 함수는 한 장치상황으로부터 다른 장치상황에 비트매프를 복사한다. 아래에 선언을 보여 주었다.

```
BOOL BitBlt(HDC hDest, int X, int Y, int Width, int Height,
            HDC hSource, int SourceX, int SourceY, DWORD dwHow);
```

hDest 에는 복사측의 장치상황을 설정하고 X 및 Y 에는 비트매프를 그리는 자리표를 설정한다. 복사측의 령역의 너비와 높이는 Width 및 Height 에 설정한다. hSource 에는 복사원천으로 되는 장치상황의 손잡이를 설정한다. 여기에서는 이것이 CreateCompatibleDC( )에서 얻은 기억기장치상황으로 된다.

SourceX 및 SourceY 에는 복사측의 비트매프의 왼쪽웃모서리의 자리표를 설정한다. dwHow 에는 비트매프를 화면에 어떻게 그리는가를 설정한다. 자주 사용되는 dwHow 의 설정값(마크로)을 아래에 보여 주었다.

마 크 로	효　　과
DSTINVERT	전송측의 비트매프를 반전한다.
SRCAND	전송원과 전송측의 비트매프로 AND 연산을 진행한다.
SRCCOPY	전송원의 비트매프를 복사하고 전송측에 덧쓰기한다.
SRCPAINT	전송원과 전송측의 비트매프로 OR 연산을 진행한다.
SRCINVERT	전송원과 전송측의 비트매프로 XOR 연산을 진행한다.

BitBlt( )는 호출이 성공하면 령 아닌 값을 돌려 주며 실패하면 령을 돌려 준다.

실례프로그람에서 BitBlt()에 대한 두개의 호출은 어느것이나 다 비트매프전체를 창문의 *의뢰자구역*에 복사하고 있다.

비트매프가 표시되면 두개의 장치상황을 해제한다. 여기에서는 BeginPaint( )로 얻은 장치상황을 해제하기 위해 EndPaint( )가 호출되고 있다.

CreateCompatibleDC( )를 리용하여 호출된 기억기장치상황을 해제하려면 DeleteDC( )를 사용하여야 한다. DeleteDC( )에 보내는 파라메터는 해제할 장치상황의 손잡이로 된다. 여기에서는 ReleaseDC( )를 사용할수 없다.(GetDC( )에서 얻은 장치상황만이 ReleaseDC( )를 사용하여 해제할수 있다.)

비트매프의 삭제

비트매프는 응용프로그람의 완료시에 삭제해야 할 자원이다. 그렇게 하기 위해서는 비트매프가 필요 없게 되였을 때 혹은 *WM_DESTROY* 통보문을 받았을 때 프로그람에서 *DeleteObject( )*를 호출하여야 한다. 아래에 *DeleteObject( )*의 선언을 보여 주었다.

```
BOOL DeleteObject(HGDIOBJ hObj);
```

hObj에 삭제할 객체(여기서는 비트매프)의 손잡이를 설정한다. 함수의 호출이 성공하면 령 아닌 값을 돌려 주며 실패하면 령이 돌려 진다.

비트매프의 완성된 실례프로그람코드

비트매프의 실례를 보여 주는 완전한 프로그람코드를 실례 7-1에 주었다.

실례 7-1. Bp 프로그람

```
// 비트매프의 표시

#include <windows.h>
#include <cstring>
#include <cstdio>

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);

char szWinName[] = "MyWin"; // 창문클라스의 이름

HBITMAP hbit;     // 비트매프의 손잡이
HINSTANCE hInst; // 실체의 손잡이

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HWND hwnd;
```

```
MSG msg;
WNDCLASSEX wcl;

// 창문클라스를 정의한다.
wcl.cbSize = sizeof(WNDCLASSEX);

wcl.hInstance = hThisInst;     // 실체의 손잡이
wcl.lpszClassName = szWinName; // 창문클라스의 이름
wcl.lpfnWndProc = WindowFunc;  // 창문함수
wcl.style = 0;                 // 체계설정형식

wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

wcl.lpszMenuName = NULL; // 클라스차림표는 없음
wcl.cbClsExtra = 0;      // 보조기억기령역은 불필요함
wcl.cbWndExtra = 0;

// 창문의 배경색은 흰색으로 한다.
wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

// 창문을 등록한다.
if(!RegisterClassEx(&wcl)) return 0;

hInst = hThisInst; // 실체의 손잡이를 보관한다.

/* 창문클라스가 등록되였으므로
    창문을 작성할수 있다. */
hwnd = CreateWindow(
  szWinName, // 창문클라스의 이름
  "Displaying a Bitmap", // 제목
  WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
  CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 너비는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 높이는 Windows가 결정하게 한다.
  NULL,          // 어미창문은 없다.
```

```
    NULL,           // 차림표는 없다.
    hThisInst,      // 실체의 손잡이
    NULL            // 추가파라메터는 없다.
  );

  // 창문을 표시한다.
  ShowWindow(hwnd, nWinMode);
  UpdateWindow(hwnd);

  // 통보문순환고리를 작성한다.
  while(GetMessage(&msg, NULL, 0, 0))
  {
    TranslateMessage(&msg); // 건반통보를 변환한다.
    DispatchMessage(&msg);  // Windows 2000 에 조종을 넘긴다.
  }

  return msg.wParam;
}

/* 이 함수는 Windows 2000 으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받아 들인다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  HDC hdc, memdc;
  PAINTSTRUCT ps;

  switch(message) {
    case WM_CREATE:
      // 비트매프의 적재
      hbit = LoadBitmap(hInst, "MyBP"); // 비트매프를 적재한다.
      break;
    case WM_PAINT:
      hdc = BeginPaint(hwnd, &ps); // 장치상황을 얻는다.

      memdc = CreateCompatibleDC(hdc); // 호환성 있는 장치상황을 얻는다.
      SelectObject(memdc, hbit); // 비트매프를 선택한다.

      BitBlt(hdc, 10, 10, 256, 128,
             memdc, 0, 0, SRCCOPY); // 화상을 표시한다.
      BitBlt(hdc, 300, 100, 256, 128,
             memdc, 0, 0, SRCCOPY); // 화상을 표시한다.
```

```
        EndPaint(hwnd, &ps); // 장치상황을 해제한다.
        DeleteDC(memdc); // 기억기장치상황을 해제한다.
        break;
      case WM_DESTROY: // 프로그람을 끝낸다.
        DeleteObject(hbit); // 비트매프를 삭제한다.
        PostQuitMessage(0);
        break;
      default:
        /* 이 switch 문에서 지정된것 이외의 통보문은
           Windows 2000 이 처리하게 한다. */
        return DefWindowProc(hwnd, message, wParam, lParam);
    }

    return 0;
}
```

프로그람의 실행결과는 그림 7-2 에 보여 주었다.

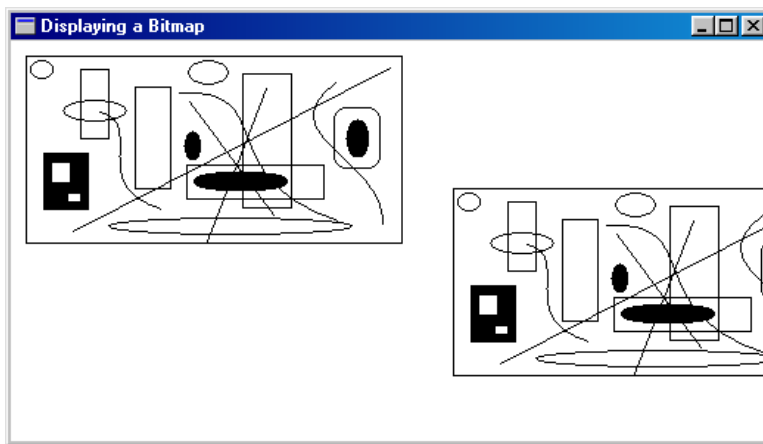



그림 7-2. 비트매프 프로그람의 실행결과

이 실례프로그람을 개조하여 여러가지 실험을 해보는것이 유익하다. 실례로 BitBlt( )의 파라메터들을 변경해 보는것 등이다. 다른 크기의 비트매프 등도 실제로 시험해 보는것이 좋다.

다시 한보 전진

창문과 화상의 XOR 연산

이 장에서 설명한것처럼 *BitBlt( )*로는 여러가지 방법으로 한 장치상황으로부터 다른 장치상황에 비트매프를 복사할수 있다. 실례로 SRCPAINT 을 지정하면 화상은 전송측과 OR 연산된다. SRCAND 를 지정하면 화상이 전송측과 AND 연산된다.

이 방법들가운데서 가장 용도가 높은것은 SRCINVERT 이다. 이 방법에서는 전송원(화상)과 전송측(창문)이 XOR 연산된다. 이것이 자주 쓰이는 리유는 두가지이다.

첫번째 리유는 창문과 화상을 XOR 연산하면 화상의 표시가 보증된다는것이다. 전송원과 전송측에서 어떤 색이 사용되는가에는 관계 없이 XOR 연산된 화상은 명확하게 표시된다.

두번째 리유는 같은 화상을 같은 위치에 두번 련속 XOR 연산하면 전송측의 화상이 복원된다는것이다. 그러므로 XOR 연산은 화상을 일시적으로 표시하였다가 원래 화상을 못 쓰게 하지 않고 복귀시키는데 효과적인 방법으로 된다.

창문과 화상을 XOR 연산한 효과를 확인하려면 먼저 작성한 비트매프프로그람의 WindowFunc( )에 아래와 같은 case 문을 삽입해 보면 알수 있다.

```
case WM_LBUTTONDOWN:
        hdc = GetDC(hwnd);

        memdc = CreateCompatibleDC(hdc); // 호환성 있는 장치상황을 작성
        SelectObject(memdc, hbit); // 비트매프를 선택한다.

        // 창문과 화상을 XOR 연산한다.
        BitBlt(hdc, LOWORD(lParam), HIWORD(lParam), 256, 128,
                   memdc, 0, 0, SRCINVERT);

        ReleaseDC(hwnd, hdc);
        DeleteDC(memdc);
        break;
case WM_LBUTTONUP:
        hdc = GetDC(hwnd);

        memdc = CreateCompatibleDC(hdc); // 호환성 있는 장치상황을 작성
        SelectObject(memdc, hbit); // 비트매프를 선택한다.
```

```
// 창문과 화상으로 두번 XOR연산을 진행한다.
BitBlt(hdc, LOWORD(lParam), HIWORD(lParam), 256, 128,
                memdc, 0, 0, SRCINVERT);

ReleaseDC(hwnd, hdc);
DeleteDC(memdc);
break;
```

이 프로그람코드는 다음과 같이 동작한다. 마우스의 왼쪽 단추를 누르면 마우스지시자의 위치에서 창문과 화상이 XOR 연산되고 반전된 비트매프의 화상이 표시된다. 그대로 마우스의 왼쪽 단추를 놓으면 화상에 두번째 XOR 연산이 진행되여 비트매프가 소거되고 창문이 본래의 상태로 돌아 간다.

마우스의 왼쪽 단추를 누른 상태에서 마우스를 움직이지 않도록 주의해야 한다. 그렇지 않으면 처음과 같은 위치에서 두번째 XOR 연산이 진행되지 않기 때문에 창문이 본래의 상태로 돌아 가지 않는다.

비트매프의 동적작성

비트매프자원을 사용하기만 하는것이 아니라 프로그람에서 동적으로 비트매프를 작성할수도 있다. 비트매프자원의 내용은 미리 정의된것으로서 프로그람의 실행중에 작성되는것은 아니다.

비트매프를 *동적으로 작성*하기 위한 가장 간단한 방법은 *CreateCompatibleBitmap( )*라는 API 함수를 사용하는것이다. 이 함수는 지정된 장치상황과 호환성이 있는 장치의존비트매프를 동적으로 작성한다. 아래에 선언을 보여 주었다.

```
HBITMAP CreateCompatibleBitmap(HDC hdc, int width, int height);
```

hdc 는 호환성 있는 비트매프를 작성하는 장치상황의 손잡이이다. 비트매프의 크기는 width와 height로 설정된다. 이것들의 단위는 화소이다. 이 함수는 호환성 있는 비트매프의 손잡이를 돌려 주며 호출이 실패한 경우는 NULL 을 돌려 준다. 초기에 비트매프는 비여 있다.

동적으로 작성된 비트매프의 사용법

비트매프를 필요에 따라 동적으로 작성할수 있는 기능은 여러가지 측면에서 활용할수 있다. 동적으로 작성된 비트매프를 사용하여 실행시에 화상을 구축할수 있다. 동적으로 작성된 비트매프에 거대한 비트매프의 화상 또는 그의 일부를 일시적으로 보관할수도 있다.

동적으로 작성된 비트매프의 가장 편리한 사용방법의 하나로서 창문의 의뢰자구역의 내용을 비트매프에 복사하는것이 있다. 이에 의해 창문의 내용을 간단히 보관할수 있다.

동적으로 작성된 비트매프의 사용방법을 익히기 위해 간단한 실례프로그람을 작성해보자. 이 프로그람은 창문의 일부 령역을 다른 령역에 복사하는 프로그람이다. 프로그람은 다음과 같이 동작한다. 우선 마우스의 왼쪽 단추를 누르면 마우스의 현재위치를 왼쪽웃모서리로 하여 창문의 100×100화소의 령역이 동적으로 작성된 비트매프에 복사된다. 다음 마우스의 오른쪽 단추를 누르면 창문에서 마우스의 현재위치에 비트매프가 복사된다. 다시말하여 기본창문의 일부를 의뢰자구역에 임의로 복사한다.

동적비트매프프로그람의 전체 코드를 실례 7-2에 보여 주었다.

실례 7-2. Dynamic 프로그람

```
/* 이 프로그람은 동적으로 비트매프를 작성하고
기본창문의 의뢰자구역에 부분적으로 복사한다. */

#include <windows.h>
#include <cstring>
#include <cstdio>

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);

char szWinName[] = "MyWin"; // 창문클라스의 이름

HINSTANCE hInst; // 실체의 손잡이
HBITMAP hdynbit; // 동적으로 작성된 비트매프의 손잡이

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;

  // 창문클라스를 정의한다.
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst;      // 실체의 손잡이
  wcl.lpszClassName = szWinName; // 창문클라스의 이름
  wcl.lpfnWndProc = WindowFunc;  // 창문함수
  wcl.style = 0;                 // 체계설정형식
```

```
    wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
    wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
    wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

    wcl.lpszMenuName = NULL; // 클라스차림표는 없음
    wcl.cbClsExtra = 0;       // 보조기억기령역은 불필요함
    wcl.cbWndExtra = 0;

    // 창문의 배경색은 흰색으로 한다.
    wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

    // 창문클라스를 등록한다.
    if(!RegisterClassEx(&wcl)) return 0;

    hInst = hThisInst; // 실체의 손잡이를 보관한다.

    /* 창문클라스가 등록되였으므로
      창문을 작성할수 있다. */
    hwnd = CreateWindow(
      szWinName, // 창문클라스의 이름
      "Using Dynamic Bitmaps", // 제목
      WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
      CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
      CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
      CW_USEDEFAULT, // 너비는 Windows가 결정하게 한다.
      CW_USEDEFAULT, // 높이는 Windows가 결정하게 한다.
      NULL,          // 어미창문은 없다.
      NULL,          // 차림표는 없다.
      hThisInst,     // 실체의 손잡이
      NULL           // 추가파라메터는 없다.
    );

    // 창문을 표시한다.
    ShowWindow(hwnd, nWinMode);
    UpdateWindow(hwnd);

    // 통보문순환고리를 작성한다.
    while(GetMessage(&msg, NULL, 0, 0))
    {
      TranslateMessage(&msg); // 건반통보를 변환한다.
      DispatchMessage(&msg);  // Windows 2000에 조종을 넘긴다.
    }

    return msg.wParam;
  }

  /* 이 함수는 Windows 2000으로부터 호출되여
     통보문대기렬에서 꺼낸 통보문을 받아 들인다.
  */
  LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                              WPARAM wParam, LPARAM lParam)
  {
```

```
    HDC hdc, memdc;
    PAINTSTRUCT ps;
    int i;
    char str[80] = "Using a dynamic bitmap.";
    char instructions[] =  "Click left button at source, "
                           "right button at destination.";

    switch(message) {
      case WM_CREATE:
        // 비트매프를 동적으로 작성한다.
        hdc = GetDC(hwnd);
        hdynbit = CreateCompatibleBitmap(hdc, 100, 100);
        ReleaseDC(hwnd, hdc);
        break;
      case WM_LBUTTONDOWN: // 창문의 일부를 비트매프에 복사한다.
        hdc = GetDC(hwnd);

        memdc = CreateCompatibleDC(hdc); // 호환성 있는 장치상황을 작성한다.

        // 동적비트매프를 기억기장치상황에 선택한다.
        SelectObject(memdc, hdynbit);

        // 령역을 동적비트매프에 복사한다.
        BitBlt(memdc, 0, 0, 100, 100,
               hdc, LOWORD(lParam), HIWORD(lParam), SRCCOPY);

        ReleaseDC(hwnd, hdc);
        DeleteDC(memdc);
        break;
      case WM_RBUTTONDOWN: // 비트매프를 창문에 복사한다.
        hdc = GetDC(hwnd);

        memdc = CreateCompatibleDC(hdc); // 호환성 있는 장치상황을 작성한다.

        // 동적비트매프를 기억기장치상황에 선택한다.
        SelectObject(memdc, hdynbit);


        // 동적비트매프를 창문에 복사한다.
        BitBlt(hdc, LOWORD(lParam), HIWORD(lParam), 100, 100,
               memdc, 0, 0, SRCCOPY);


        ReleaseDC(hwnd, hdc);
        DeleteDC(memdc);
        break;
      case WM_PAINT:
        hdc = BeginPaint(hwnd, &ps); // 장치상황을 얻는다.
```

```
        TextOut(hdc, 1, 0, instructions, strlen(instructions));

        for(i=1; i<10; i++)
          TextOut(hdc, 1, i*20, str, strlen(str));

        EndPaint(hwnd, &ps); // 장치상황을 해제한다.
        break;
      case WM_DESTROY: // 프로그람을 끝낸다.
        DeleteObject(hdynbit); // 비트매프를 삭제한다.
        PostQuitMessage(0);
        break;
      default:
        /* 이 switch 문에서 지정되지 않은 통보문들은
           Windows 2000 에 처리를 맡긴다. */
        return DefWindowProc(hwnd, message, wParam, lParam);
    }

    return 0;
  }
```

프로그람의 실행결과는 그림 7-3 에 보여 주었다.

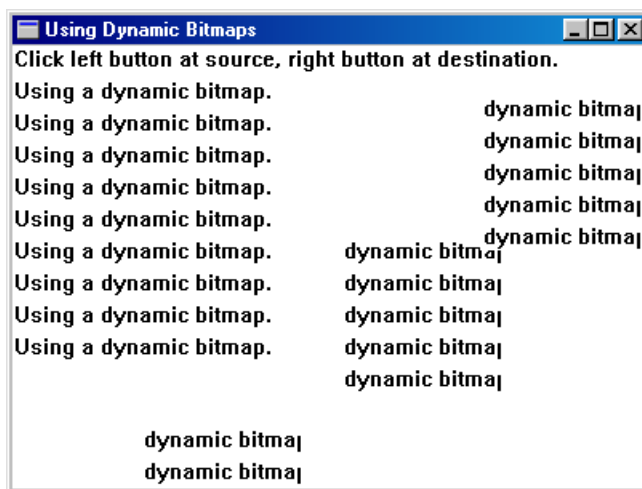



그림 7-3. 동적비트매프프로그람의 실행결과

프로그람이 실행하고 기본창문이 WM_CREATE 통보문을 받았을 때 기본창문에서

사용되는 장치상황과 호환성 있는 비트매프가 작성된다. 이 비트매프의 손잡이는 대역변수 hdynbit 에 보관된다.

마우스의 왼쪽 단추가 눌리우면 호환성 있는 기억기장치상황이 작성되고 그 장치상황에 비트매프가 선택되여 창문의 100 × 100 화소의 4 각형령역이 비트매프에 복사된다. 마우스의 오른쪽 단추가 눌리우면 같은 순서로 처리가 진행되는데 전송측과 전송원은 반대로 된다. 이런 절차에 의하여 비트매프의 내용이 창문에 복사된다.

동적으로 비트매프를 작성하는 기능에는 여러가지 활용방법이 있다.

다시그리기문제의 해결방법

비트매프를 취급하는데 필요되는 기본적인 지식을 습득하였으므로 이제는 그 지식을 활용하여 Windows 2000 프로그람작성에 있어서의 가장 기본적인 문제를 해결하기로 하자. 이 문제는 WM_PAINT 통보문을 받았을 때 창문의 내용을 다시 그리는것이다.

Windows 2000(및 모든 판본의 Windows)에서는 WM_PAINT 통보문을 받을 때마다 창문의 의뢰자구역의 내용을 다시그리기하는것이 응용프로그람의 역할로 되여 있다.

일반적으로 창문이 재표시될 때는 그 내용을 다시 그려야 한다. 프로그람을 실행하고 창문의 의뢰자구역에 어떤 출력을 진행한 다음 그것이 다른 창문의 밑에 가리웠다면 출력한 내용이 없어 진다. 그때문에 창문이 재표시될 때 그 내용을 다시그리기할 필요가 있는것이다. 비트매프의 표시에서 리용한 기술을 이 다시그리기문제를 해결하는데 사용할수 있다.

먼저 창문을 다시그리기하기 위한 세가지 기본적인 방법을 복습하자.

첫번째 방법은 어떤 계산에 의해 출력된 내용을 다시 작성하고 변경하는것이다.

두번째 방법은 표시에 관계되는 사건을 기록해 두고 그 사건들을 재생하는것이다.

그리고 세번째 방법은 *가상창문*을 작성하고 WM_PAINT 통보문을 받을 때마다 가상창문의 내용을 실제 창문에 단순히 복사하는것이다. 이가운데서 가장 일반적인 방법은 물론 세번째 방법이다. 실제로 프로그람을 작성해 보자.

가상창문을 사용하여 다시그리기를 진행하는 순서는 다음과 같다.

우선 응용프로그람에서 사용되고 있는 실제의 장치상황과 호환성이 있는 기억기장치상황 및 비트매프(이것이 가상창문으로 된다.)를 작성한다.

다시말하여 가상창문이란 프로그람의 실제창문과 호환성을 가지는 비트매프인것이다. 가상창문을 작성한 다음에는 기본창문의 의뢰자구역에 출력되는 모든 내용을 가상창문에도 써넣어야 한다.

이렇게 하면 가상창문의 내용은 실제창문의 내용의 완전한 복사판으로 된다. WM_PAINT 통보문을 받으면 가상창문의 내용을 실제창문에 복사하여 창문의 내용을 다시그리기한다. 이렇게 일단 가리워 있다가 다시 표시된 창문이 WM_PAINT 통보문을

받았을 때 그 내용이 자동적으로 다시그리기된다.

필요한 API 함수

가상창문을 실현하는데 몇가지 API 함수가 필요된다. 그중 다섯개는 이미 설명한것으로서 *CreateCompatibleBitmap( )*, *CreateCompatibleDC( )*, *SelectObject( )*, *GetStockObject( )* 및 *BitBlt( )*이다. 그밖에 *PatBlt( )*와 *GetSystemMetrics( )*가 필요되므로 여기에서 이 함수들의 기능을 설명을 하기로하자.

PatBlt( )

PatBlt( )함수는 현재 선택되여 있는 *붓*의 색과 무늬로 4각형령역을 채운다. 붓이란 창문 또는 령역을 채우기 위한 객체이다. 붓을 사용하여 령역을 채운다는것은 령역을 색칠한다는것을 의미한다. 아래에 *PatBlt( )*의 선언을 보여 주었다.

```
BOOL PatBlt(HDC hdc, int X, int Y, int width, int height,
                              DWORD dwHow);
```

hdc 는 색칠을 진행하는 장치의 손잡이이다. X 와 Y 에 색칠하는 령역의 왼쪽웃모서리의 자리표를 설정한다. 령역의 너비와 높이를 width 와 height 에 설정한다. dwHow 의 값은 붓의 사용방법을 지정하며 아래의 마크로들중에서 어느 하나의 값을 설정한다.

마크로	의　　미
BLACKNESS	령역이 무색으로 된다.(붓은 무시된다.)
WHITENESS	령역이 백색으로 된다.(붓은 무시된다.)
PATCOPY	붓이 령역에 복사된다.
PATINVERT	붓과 령역이 OR 연산된다.
DSTINVERT	령역이 반전된다.(붓은 무시된다.)

붓을 그대로 사용하려면 dwHow 에 *PATCOPY* 를 설정하면 된다. 호출이 성공하면 PatBlt( )는 령 아닌 값을 돌려 주며 실패하면 령을 돌려 준다.

체계치수를 얻기

가상창문을 작성하려면 가상창문에서 리용되는 비트매프의 크기를 알아야 할 필요가 있다. 그를 위한 방법의 하나로서 화면의 크기를 화소단위로 얻고 그것을 비트매프의 크기로 할수 있다. 화면크기 등의 정보를 얻으려면 *GetSystemMetrics( )*라는 API 함수를 사용한다. 아래에 선언을 보여 주었다.

```
int GetSystemMetrics(int what);
```

what 은 얻으려는 정보를 지정한다. GetSystemMetrics( )을 사용하면 여러가지 정보를 얻을수 있다. 아래에 what 에 지정할수 있는 주요한 값들을 표시한다.

값	얻어 지는 치수정보
SM_CXFULLSCREEN	최대화되였을 때의 의뢰자구역의 너비
SM_CYFULLSCREEN	최대화되였을 때의 의뢰자구역의 높이
SM_CXICON	표준적인 아이콘의 너비
SM_CYICON	표준적인 아이콘의 높이
SM_CXSMICON	작은 아이콘의 너비
SM_CYSMICON	작은 아이콘의 높이
SM_CXSCREEN	화면전체의 너비
SM_CYSCREEN	화면전체의 높이

여기에서는 가상창문의 크기를 결정하기 위해 *SM_CXSCREEN* 과 *SM_CYSCREEN* 을 사용하기로 한다. 이 마크로들을 지정하여 얻어지는 값의 단위는 화소이다.

가상창문의 작성과 사용방법

이제부터 작성하게 될 프로그람의 내용에 대해 간단히 설명한다.

WM_PAINT 통보문을 받았을 때 창문을 다시그리기하는 간단하고 편리한 방법으로 가상창문을 작성한다. 창문의 의뢰자구역에 출력되는 내용은 *물리창문*(실제 창문)과 가상창문(동적으로 작성된 비트매프)의 량측에 써넣기된다. 다시그리기요구를 받을 때마다 가상창문의 내용이 실제로 화면에 표시되여 있는 물리창문에 복사된다. 이에 의해 창문의 내용이 다시그리기된다.

가상창문을 작성하기 위해 제일 먼저 진행해야 하는 처리는 실제의 장치상황과 호환성이 있는 기억기장치상황을 얻는것이다. 이것은 창문이 처음 작성되였을 때 한번만 진행하는 처리이다. 호환성 있는 장치상황은 프로그람이 실행하는 동안 보관된다. 아래에 이 기능을 실현하는 프로그람코드를 보여 준다.

```
case WM_CREATE:
            // 화면의 크기를 얻는다.
            maxX = GetSystemMetrics(SM_CXSCREEN);
            maxY = GetSystemMetrics(SM_CYSCREEN);

            // 호환성 있는 기억기화상을 작성한다.
            hdc = GetDC(hwnd);
            memdc = CreateCompatibleDC(hdc);
```

```
            hbit = CreateCompatibleBitmap(hdc, maxX, maxY);
            SelectObject(memdc, hbit);
            hbrush = GetStockObject(WHITE_BRUSH);
            SelectObject(memdc, hbrush);
            PatBlt(memdc, 0, 0, maxX, maxY, PATCOPY);
            ReleaseDC(hwnd, hdc);
            break;
```

프로그람코드의 내용을 구체적으로 따져 보자. 먼저 화면의 크기를 얻는데 이 값은 호환성 있는 비트매프를 작성하는데 쓰인다. 다음 현재의 장치상황을 얻는다. 그리고 CreateCompatibleDC( )를 호출하여 호환성 있는 장치상황을 기억기에 작성한다. 이 장치상황의 손잡이는 대역변수 memdc에 보관된다.

다음 호환성 있는 비트매프를 작성한다. 이것은 가상창문과 물리창문에 1:1 대응된다. 비트매프의 크기가 화면의 최대 크기로 되여 있으므로 물리적창문의 크기가 어떤 크기라고 해도 비트매프는 창문을 다시그리기하는데 충분한 크기로 된다.(실제로는 약간 작은 크기로 해도 상관 없다. 왜냐하면 창문의 경계선을 다시그리기할 필요는 없기때문이다.) 비트매프의 손잡이는 대역변수 hbit 에 보관된다. 이것을 기억기장치상황에 선택한다.

다음 흰색의 붓을 선택하고 그의 손잡이를 대역변수 hbrush 에 보관한다. 이 붓을 기억기장치상황에 선택하고 *PatBlt( )*를 호출한 다음 가상창문전체를 색칠한다. 이에 의해 가상창문의 배경이 흰색으로 되며 다음에 작성하는 실례프로그람의 물리적창문의 배경색과 같이 된다.

마지막으로 물리창문의 장치상황이 해제된다. 그러나 기억기장치상황은 프로그람을 완료할 때까지 해제되지 않는다.

가상창문을 작성하면 반드시 모든 출력을 가상창문에도 동시에 해야 한다. WM_PAINT 통보문을 받을 때마다 물리창문의 내용을 다시그리기하는데 가상창문의 내용을 리용한다. 이것을 실현하는 방법은 앞절에서 BitBlt( )함수를 사용하여 창문에 비트매프의 화상을 복사하는것과 같다.

가상창문의 실례를 보여 주는 전체 프로그람코드

다시그리기 문제를 해결하기 위하여 가상창문을 사용하는 실례를 보여 주는 완전한 프로그람코드를 실례 7-3에 보여 주었다.

이 프로그람은 WM_CHAR 통보문에 응답하여 가상창문과 물리창문의 량측에 문자렬을 출력한다.(이것은 건입력된 문자렬을 물리창문과 가상창문의 량측에 표시한다는것이다.) WM_PAINT 통보문을 받았을 때 물리창문을 다시그리기하기 위해 가상창문의 내용을 리용한다.

실례 7-3. Repaint 프로그람

```
// 가상창문을 리용한 다시그리기

#include <windows.h>
#include <cstring>
#include <cstdio>

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);

char szWinName[] = "MyWin"; // 창문클라스의 이름

char str[255]; // 출력할 문자렬을 보관한다.

int X=0, Y=0;   // 현재 출력위치
int maxX, maxY; // 화면의 크기

HDC memdc;      // 가상장치손잡이를 보관한다.
HBITMAP hbit;  // 가상비트매프를 보관한다.
HBRUSH hbrush; // 붓의 손잡이를 보관한다.

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;

  // 창문클라스를 정의한다.
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst;      // 실체의 손잡이
  wcl.lpszClassName = szWinName; // 창문클라스의 이름
  wcl.lpfnWndProc = WindowFunc;  // 창문함수
  wcl.style = 0;                 // 체계설정의 형식

  wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
  wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
```

```
    wcl.hCursor = LoadCursor(NULL, IDC_ARROW); // 유표의 형식

    wcl.lpszMenuName = NULL; // 클라스차림표는 없다.
    wcl.cbClsExtra = 0;      // 보조기억기령역은 불필요함
    wcl.cbWndExtra = 0;

    // 창문의 배경색을 흰색으로 한다.
    wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

    // 창문클라스를 등록한다.
    if(!RegisterClassEx(&wcl)) return 0;

    /* 창문클라스가 등록되였으므로
        창문을 작성할수 있다. */
    hwnd = CreateWindow(
      szWinName, // 창문클라스의 이름
      "Using a Virtual Window", // 제목
      WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
      CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
      CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
      CW_USEDEFAULT, // 너비는 Windows가 결정하게 한다.
      CW_USEDEFAULT, // 높이는 Windows가 결정하게 한다.
      NULL,          // 어미창문은 없다.
      NULL,          // 차림표는 없다.
      hThisInst,     // 실체의 손잡이
      NULL           // 추가파라메터는 없다.
    );

    // 창문을 표시한다.
    ShowWindow(hwnd, nWinMode);
    UpdateWindow(hwnd);

    // 통보문순환고리를 작성한다.
    while(GetMessage(&msg, NULL, 0, 0))
    {
      TranslateMessage(&msg); // 건반통보를 변환한다.
      DispatchMessage(&msg);  // Windows 2000에 조종을 넘긴다.
    }

    return msg.wParam;
  }
```

```
/* 이 함수는 Windows 2000으로부터 호출되여
통보문대기렬에서 꺼낸 통보문을 받는다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  HDC hdc;
  PAINTSTRUCT ps;

  switch(message) {
    case WM_CREATE:
      // 화면의 크기를 얻는다.
      maxX = GetSystemMetrics(SM_CXSCREEN);
      maxY = GetSystemMetrics(SM_CYSCREEN);

      // 호환성 있는 기억기화상을 작성한다.
      hdc = GetDC(hwnd);
      memdc = CreateCompatibleDC(hdc);
      hbit = CreateCompatibleBitmap(hdc, maxX, maxY);
      SelectObject(memdc, hbit);
      hbrush = (HBRUSH) GetStockObject(WHITE_BRUSH);
      SelectObject(memdc, hbrush);
      PatBlt(memdc, 0, 0, maxX, maxY, PATCOPY);
      ReleaseDC(hwnd, hdc);
      break;
    case WM_CHAR:
      hdc = GetDC(hwnd);
      sprintf(str, "%c", (char) wParam); // 문자를 문자렬로 한다.

      // 복귀, 개행의 간단한 처리
      if((char)wParam == '\r') {
        Y += 16; // 개행한다.
        X = 0;   // 복귀한다.
      }
      else {
        TextOut(memdc, X, Y, str,
                strlen(str)); // 기억기에 출력한다.
        TextOut(hdc, X, Y, str,
```

```
                    strlen(str)); // 창문에 출력한다.
            X += 10;
        }
        ReleaseDC(hwnd, hdc);
        break;
    case WM_PAINT: // 다시그리기요구를 처리한다.
        hdc = BeginPaint(hwnd, &ps); // 장치상황을 얻는다.

        // 기억기화상을 화면에 복사한다.
        BitBlt(hdc, 0, 0, maxX, maxY, memdc, 0, 0, SRCCOPY);
        EndPaint(hwnd, &ps); // 장치상황을 해제한다.
        break;
    case WM_DESTROY: // 프로그람을 완료한다.
        DeleteDC(memdc); // 기억기장치상황을 해제한다.
        DeleteObject(hbit); // 비트매프를 삭제한다.
        PostQuitMessage(0);
        break;
    default:
        /* 이 switch 문에서 지정되지 않은 통보문들은
           Windows 2000 이 처리하게 한다. */
      return DefWindowProc(hwnd, message, wParam, lParam);
  }

  return 0;
}
```

프로그람의 실행결과를 그림 7-4 에 보여 주었다.

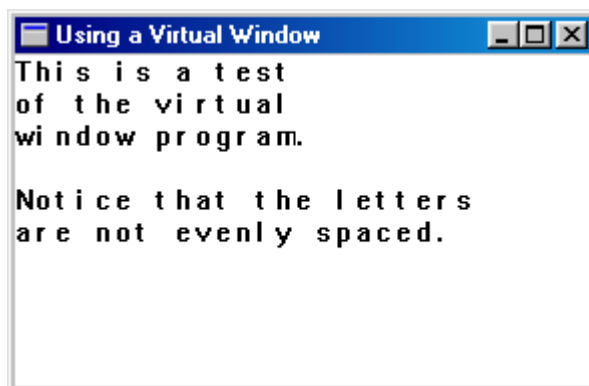



그림 7-4. 가상창문 프로그람의 실행결과

프로그람의 내용을 상세히 보자. 먼저 WM_CHAR 통보문과 관련된 프로그람코드를 설명한다.

```
case WM_CHAR:
  hdc = GetDC(hwnd);
  sprintf(str, "%c", (char) wParam); // 문자를 문자렬로 한다.

  // 복귀, 개행의 간단한 처리
  if((char)wParam == '\r') {
    Y += 16; // 개행한다.
    X = 0;   // 복귀한다.
  }
  else {
    TextOut(memdc, X, Y, str,
            strlen(str)); // 기억기에 출력한다.
    TextOut(hdc, X, Y, str,
            strlen(str)); // 창문에 출력한다.
    X += 10;
  }
  ReleaseDC(hwnd, hdc);
  break;
```

웅근수형의 대역변수 X 와 Y 는 출력할 문자의 위치를 지정하는데 사용된다. 이 변수들의 값은 령으로 초기화되여 있다. 문자가 입력되면 문자가 문자렬로 변환되고 물리창문과 가상창문의 량쪽에 출력된다. 이렇게 되여 건입력의 완전한 복사가 가상창문에도 써넣어 진다.

한 문자의 입력에 대해 X 의 값이 10 씩 증가하며 [Enter]건이 눌리우면 Y 의 값이 16 만큼 증가하여 개행이 진행된다는것을 알수 있다. 이것은 X 와 Y 의 위치를 결정하기 위한 간단한 방법이다. 다음 장에서는 문자렬을 가장 정확하게 출력하는 방법을 설명한다.

WM_PAINT 통보문을 받을 때마다 가상창문의 내용이 물리창문에 복사되여 그의 내용이 다시그리기된다. 이것은 아래의 프로그람코드에서 실현되고 있다.

```
case WM_PAINT: // 다시그리기요구를 처리한다.
  hdc = BeginPaint(hwnd, &ps); // 장치상황을 얻는다.

  // 기억기화상을 화면에 복사한다.
  BitBlt(hdc, 0, 0, maxX, maxY, memdc, 0, 0, SRCCOPY);
  EndPaint(hwnd, &ps); // 장치상황을 해제한다.
break;
```

BitBlt( )함수는 memdc 로부터 hdc 에 화상을 복사한다. *SRCCOPY* 파라메터는 전송원의 화상을 변화시키지 않고 전송측에 복사할것을 지정한다. 모든 출력이 memdc 에 보관되여 있으므로 이 처리를 진행하여 창문을 다시그리기할수 있다. 어떤 문자를 입력한 다음 창문을 다른 창문에 가리우게 하였다가 다시 표시하면 입력된 문자가 다시그리기되는것을 학인할수 있다.

가상창문을 사용하여 다른 방법으로도 다시그리기를 진행할수 있다. 정보를 두번 출력하지 않고 물리창문과 가상창문의 어느 한 측에 출력하는 경우 건입력시에는 출력창문에만 출력을 하고 거기서 InvalidateRect( )를 호출하면 물리창문에도 출력할수 있다.(제 3 장에서 설명한것처럼 InvalidateRect( )는 WM_PAINT 통보문을 생성한다.)

그러나 이 방법을 리용하면 건입력을 진행할 때마다 창문이 다시그리기되므로 창문이 약간 흐르는것처럼 보이게 된다. 그러므로 여기서 작성한 실례프로그람에서는 이 방법이 적합치 못하다. 하지만 출력이 거의나 변화되지 않는 프로그람이라면 이 방법도 적용할수 있다.

다시그리기기능의 개선

지금까지 설명한것은 가상창문을 리용하여 다시그리기를 진행하는 일반적인 방법이였다. 하지만 프로그람을 보다 간단하게 개량할수도 있다.

앞에서 본 프로그람에는 두가지 문제가 있다. 첫번째 문제는 물리창문의 크기에 관계 없이 가상창문전체가 물리창문에 복사된다는것이다. 가상창문의 크기가 화면전체의 크기와 같으므로 이 방법에는 명백히 결함이 있다. 물론 이렇게 해도 표시상에서는 문제가 없지만 CPU 시간을 랑비하게 된다.

두번째 문제는 물리창문에서 다시그리기할 령역의 크기에 관계 없이 WM_PAINT 통보문을 받을 때마다 창문전체를 다시그리기하는것이다. 이것은 Windows 프로그람작성경험이 적은 사람들이 사용하는 방법이며 물론 최량의 방법은 아니다.

실제로 다시그리기가 필요로 되는것은 대체로 창문의 일부분만이다. 창문을 다시그리기하는데는 많은 시간이 걸린다는것을 잊어서는 안된다. 창문이 크면 클수록 다시그리기에는 긴 시간이 소모된다. 창문에서 다시그리기가 불필요한 부분까지 다시그리기하는것은 시간을 랑비하고 응용프로그람의 성능을 저하시킨다.

창문에서 실제로 다시그리기해야 할 부분만을 다시그리기하도록 프로그람을 고치면 다시그리기의 시간이 단축되고 응용프로그람은 빠른 속도로 동작하게 된다. 프로그람의 성능을 향상시키려면 창문의 다시그리기를 최량화하는것이가장 효과적이다.

Windows 2000 은 보다 효률적인 다시그리기처리를 실현하는데 필요되는 정보를 BeginPaint( )에서 제공해 준다.

BeginPaint( )의 상세

제 3 장에서 설명한것처럼 WM_PAINT 통보문을 처리할 때는 *BeginPaint( )*를 호출하여 장치상황을 얻어야 한다. 장치상황을 얻는것외에도 BeginPaint( )에서는 창문의 표시상태에 관계되는 정보도 얻을수 있다. 이 정보를 사용하여 다시그리기를 최적화할수 있다.

아래에 다시 한번 BeginPaint( )의 선언을 보여 주었다.

```
HDC BeginPaint(HWND hwnd, PAINTSTRUCT *lpPS);
```

BeginPaint( )의 호출이 성공하면 장치상황의 손잡이를 돌려 주며 실패하면 NULL을 돌려 준다. hwnd 는 장치상황을 얻으려는 창문의 손잡이이다. 두번째 파라메터는 *PAINTSTRUCT 구조체*의 지시자이다. lpPS에서 지정된 구조체는 창문을 다시그리기하는데 리용되는 정보를 가지고 있다. PAINTSTRUCT 의 정의를 아래에 보여 주었다.

```
typedef struct tagPAINTSTRUCT {
    HDC hdc;                        // 장치상황의 손잡이
    BOOL fErase;                    // 배경을 소거해야 하는 경우는 TRUE
    RECT rePaint;                   // 다시그리기해야 하는 령역의 자리표
    BOOL fRestore;                  // Windows에 예약된 성원
    BOOL fIncUpdate;                // Windows에 예약된 성원
    BYTE rgbReserved[32];           // Windows에 예약된 성원
}PAINTSTRUCT;
```

여기서 특히 주목해야 할 성원은 rcPaint 이다. 이 성원에는 다시그리기할 필요가 있는 창문 령역의 자리표가 보관되여 있다. 이 정보를 리용하여 창문을 다시그리기하는 시간을 단축할수 있다.

가상창문의 다시그리기시간의 단축

시간단축의 요점은 WM_PAINT 통보문을 받았을 때 rcPaint 가 가리키는 창문의 일부 령역만을 다시 그리는것이다. 가상창문을 사용하면 이것을 간단히 실현할수 있다. 가상창문의 같은 령역을 물리창문에 복사하기만 하면 되기때문이다.

두개의 장치상황은 동등한것이므로 자리표에 대해서도 마찬가지이다. rcPaint 에서 지정되는 자리표는 물리창문과 가상창문의 어느것에서나 사용할수 있다. 실례로 WM_PAINT 통보문에 대한 응답을 아래와 같이 한다.

```
case WM_PAINT:  // 다시그리기요구의 처리(개량판)
            hdc = BeginPaint(hwnd, &ps); // 장치상황을 얻는다.

            // 가상창문의 일부를 복사한다.
            BitBlt(hdc, ps.rcPaint.left, ps.rcPaint.top,
            ps.rcPaint.right - ps.rcPaint.left, // 너비
            ps.rcPaint.bottom - ps.rcPaint.top,
            memdc,
            ps.rcPaint.left, ps.rcPaint.top,
            SRCCOPY);

    EndPaint(hwnd, &ps);  // 장치상황을 해제한다.
    break;
```

고친 효과를 확인하기 위해 이 프로그람코드를 앞절에서 작성한 프로그람의 해당한 부분과 치환한다. 개량된 프로그람코드에서는 rcPaint 에서 지정된 령역만을 복사하므로 정보의 덧쓰기나 창문의 경계선을 벗어난 그리기가 진행되지 않으므로 시간을 랑비하지 않는다.

가상창문을 사용한 혜택으로 창문의 *다시그리기의 최적화*가 간단히 실현되였기때문이다. 다시그리기에 가상창문을 리용하는 방법은 다시그리기와 관련된 많은 처리에서 간결한 해결책을 제공해 준다.

실험으로서 BeginPaint( )를 호출한 다음에 아래와 같은 프로그람코드를 삽입하면 WM_PAINT 통보문을 받았을 때 필요한 다시그리기령역의 자리표를 확인할수 있다.

```
sprintf(str, "top, left: &d &d\nBottom, right: &d &d",
                ps.rcPaint.top, ps.rcPaint.left,
                ps.rcPaint.bottom, ps.rcPaint.right);
MessageBox(hwnd, str, "coordinates", MB_OK);
```

str 는 문자의 배렬이다. 프로그람을 실행하고 창문의 두개 개소 혹은 그이상의 부분을 다른 창문에 가리우게 한다. 매 가리운 부분에 대해 통보문이 생성된다는것을 알수 있을것이다.

이 책의 대부분의 실례프로그람에서는 다시그리기의 최적화를 하지 않고 있다. 왜냐

하면 다시그리기를 위한 프로그람코드를 추가하면 실례프로그람에서 설명하려는 요점이 강조되지 않기때문이다. 그러나 실제로 작성하는 프로그람들에서는 WM_PAINT 통보문을 적절하게 처리하는것이 성능에 커다란 영향을 준다는것을 명심해 두어야 한다.

전용아이콘과 전용유표의 작성

이 장을 마치기에 앞서 전용아이콘과 *전용유표*를 작성하고 사용하는 방법을 설명하기로 한다.

알고 있는바와 같이 모든 Windows 2000 응용프로그람은 제일 처음 창문의 속성을 정의하는 창문클라스를 작성하며 창문클라스에는 응용프로그람의 아이콘과 마우스유표의 형태도 포함되여 있다. 아이콘과 마우스유표의 손잡이는 *WNDCLASSEX* 구조체의 hIcon, hIconSm 및 hCursor 라는 성원에 보관된다. 지금까지는 Windows 2000 이 제공하는 내장아이콘과 유표를 사용하여왔다. 하지만 아이콘과 유표를 자체로 정의할수도 있다.

아이콘과 유표의 정의

*전용아이콘*과 마우스유표를 사용하려면 그것들의 화상을 미리 화상편집기를 사용하여 작성해 두어야 한다. 작은 아이콘과 큰 아이콘이 필요된다는것을 명심해 두어야 한다. 작은 아이콘의 크기는 16×16 화소이며 큰 아이콘의 크기는 32×32 화소이다. 작은 아이콘과 큰 아이콘은 같은 아이콘파일안에 보관된다. 모든 유표의 크기는 같으며 32×32 화소이다.

이식과 관련한 요점 : 16bit 의 Windows 에서는 큰 아이콘만을 정의한다.

이제부터 작성하게 될 프로그람에서는 아이콘을 보관하는 파일을 ICON.ICO 라는 파일이름으로 작성한다. 반드시 32×32 화소와 16×16 화소의 두개의 아이콘을 작성해야 한다. 이 아이콘들은 또한 같은 파일에 보관하여야 한다.

유표를 보관하는 파일은 CURSOR.CUR 라는 이름으로 작성한다. 그림 7-5 에 이 장의 실례프로그람에서 사용하는 아이콘과 유표의 형태를 보여 주었다.

아이콘과 유표의 화상을 작성한 다음에는 프로그람의 자원파일에 *ICON* 및 *CURSOR* 의 두 명령문을 추가해야 한다. 이 명령문들의 일반적인 서식은 다음과 같다.

IconName ICON filename

CursorName CURSOR filename

IconName 은 아이콘을 식별하는 이름이며 CursorName 은 유표를 식별하는 이름이다. 이 이름들은 프로그람에서 아이콘이나 유표를 참조하는데 사용된다. filename 에는 전용아이콘이나 전용유표를 보관하고 있는 파일의 이름을 지정한다.

실례프로그람에는 아래의 명령문이 들어 있는 자원파일이 필요하다.

```
MyCursor CURSOR CURSOR.CUR
MyIcon ICON ICON.ICO
```

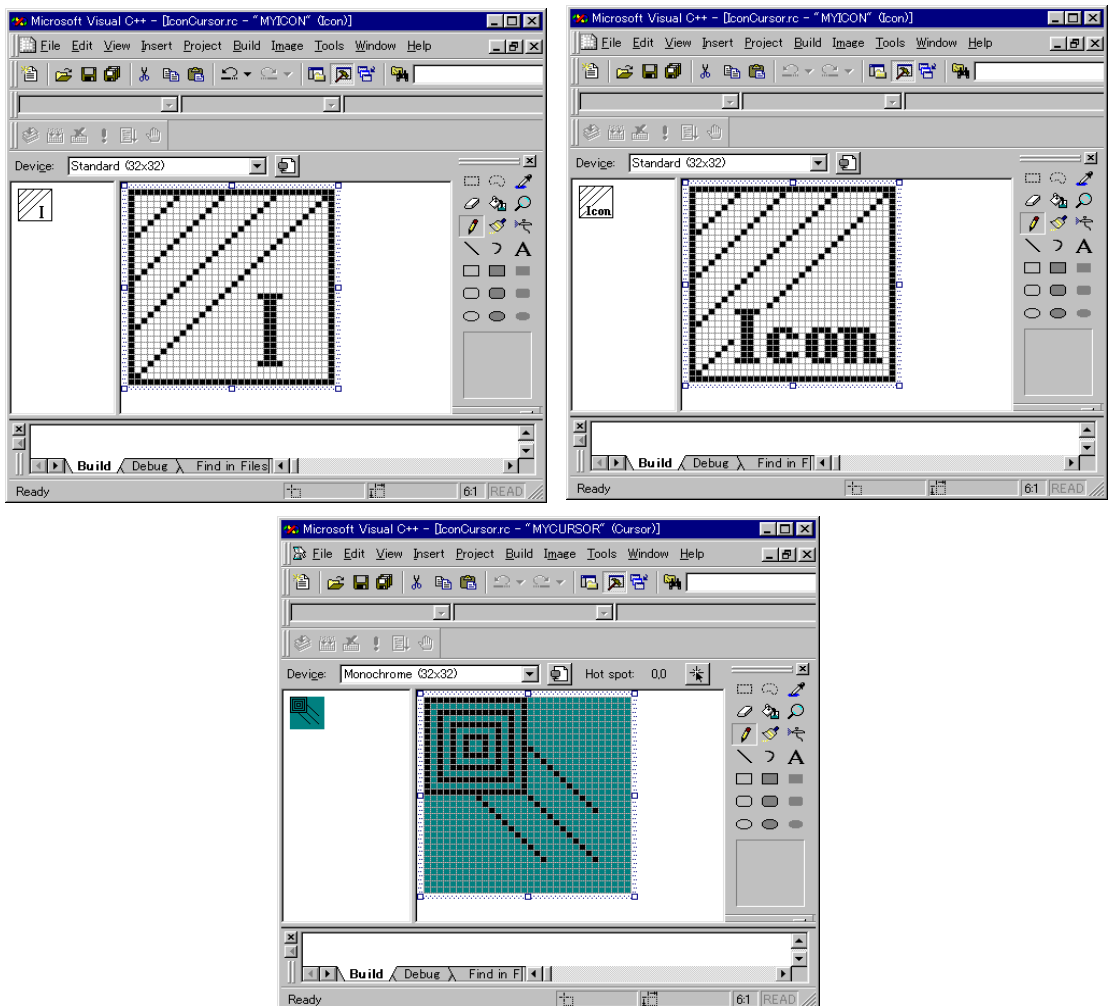



그림 7-5. 전용아이콘과 전용유표

아이콘과 유표의 적재

전용아이콘과 전용유표를 사용하려면 그것들을 창문클라스를 등록하기전에 적재하고 2 장에서 설명한 *LoadIcon( )* 및 *LoadCursor( )*라는 API 함수를 사용하여 *WNDCLASSEX* 구조체의 성원으로 설정하여야 한다.

례를 들어 아래의 프로그람코드는 MyIcon 이라는 이름의 아이콘 및 MyCursor 라는 이름의 유표를 적재하고 그것들을 WNDCLASSEX 의 성원에 설정한다.

```
wc1.hIcon = LoadIcon(hThisInst, "MyIcon");            // 큰 아이콘
wc1.hIconSm = NULL;  // MyIcon 에 보관된 작은 아이콘을 사용한다.
wc1.hCursor = LoadCursor(hThisInst, "MyCursor"); // 유표
```

hThisInst 는 프로그람의 실체의 손잡이이다. 지금까지 작성해온 프로그람들에서는 이 함수들이 체계설정의 아이콘이나 유표를 적재하는데 사용되였지만 여기에서는 전용아이콘이나 전용유표를 적재하는데 사용되고 있다.

hIconSm 에 NULL 을 설정한데 주목해야 한다. hIconSm 이 NULL 인 경우에는 큰 아이콘을 보관한 파일안에 정의되여 있는 작은 아이콘이 자동적으로 적재된다. 다시말하여 hIconSm 이 NULL 이라면 hIcon 에 설정된 큰 아이콘과 같은 파일에 보관된 작은 아이콘이 설정되게 된다.

물론 필요하다면 작은 아이콘에 다른 아이콘자원을 설정할수도 있다. 그러나 그렇게 하자면 이 장의 맨 마감에 설명하는 *LoadImage( )*함수를 사용해야 한다. 일반적인 프로그람에서는 큰 아이콘이 보관된 파일에 작은 아이콘도 정의되여 있다.

전용아이콘과 전용유표의 실례프로그람

실례 7-4 에 보여 준 프로그람은 전용아이콘과 전용유표를 사용하는 프로그람이다. 작은 아이콘은 기본창문의 체계차림표와 프로그람을 최소화하였을 때 과제띠에 표시된다. 큰 아이콘은 프로그람을 탁상면에 이동했을 때 표시된다.

전용유표는 마우스지시자를 창문우에 놓았을 때 표시된다. 프로그람의 창문우에 마우스를 이동하면 마우스유표의 형태가 프로그람에서 정의된 형태로서 자동적으로 변한다. 프로그람의 창문밖으로 마우스를 이동하면 마우스유표는 자동적으로 체계설정의 형태로 돌아간다.

이 프로그람을 번역하기전에 화상편집기를 사용하여 전용아이콘과 전용유표를 작성하고 프로그람의 자원파일에 자원정의를 추가하여야 한다.

실례 7-4. IconCursor 프로그람

```
// 전용아이콘과 전용유표의 실례
```

```
#include <windows.h>
#include <cstring>
#include <cstdio>

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);

char szWinName[] = "MyWin"; // 창문클라스의 이름

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;

  // 창문클라스를 정의한다.
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst;      // 실체의 손잡이
  wcl.lpszClassName = szWinName; // 창문클라스의 이름
  wcl.lpfnWndProc = WindowFunc;  // 창문함수
  wcl.style = 0;                  // 체계설정의 형식

  wcl.hIcon = LoadIcon(hThisInst, "MyIcon"); // 큰 아이콘
  wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
  wcl.hCursor = LoadCursor(hThisInst, "MyCursor"); // 유표

  wcl.lpszMenuName = NULL; // 클라스차림표는 없다.
  wcl.cbClsExtra = 0;      // 보조기억기령역은 필요 없다.
  wcl.cbWndExtra = 0;

  // 창문의 배경색을 흰색으로 한다.
  wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

  // 창문클라스를 등록한다.
  if(!RegisterClassEx(&wcl)) return 0;

  /* 창문클라스가 등록되였으므로
     창문을 작성할수 있다. */
  hwnd = CreateWindow(
    szWinName, // 창문클라스의 이름
    "Custom Icons and Cursor", // 제목
    WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
```

```
    CW_USEDEFAULT, // X 자리표는 Windows 가 결정하게 한다.
    CW_USEDEFAULT, // Y 자리표는 Windows 가 결정하게 한다.
    CW_USEDEFAULT, // 너비는 Windows 가 결정하게 한다.
    CW_USEDEFAULT, // 높이는 Windows 가 결정하게 한다.
    NULL,          // 어미창문은 없다.
    NULL,          // 차림표는 없다.
    hThisInst,     // 실체의 손잡이
    NULL           // 추가파라메터는 없다.
  );

  // 창문을 표시한다.
  ShowWindow(hwnd, nWinMode);
  UpdateWindow(hwnd);

  // 통보문순환고리를 작성한다.
  while(GetMessage(&msg, NULL, 0, 0))
  {
    TranslateMessage(&msg); // 건반통보를 변환한다.
    DispatchMessage(&msg);  // Windows 2000 에 조종을 넘긴다.
  }

  return msg.wParam;
}

/* 이 함수는 Windows 2000 으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받아 들인다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  switch(message) {
    case WM_DESTROY: // 프로그람을 끝낸다.
      PostQuitMessage(0);
      break;
    default:
      /* 이 switch 문에서 지정된것 이외의 통보문은
         Windows 2000 에 처리를 맡긴다. */
      return DefWindowProc(hwnd, message, wParam, lParam);
  }
```

```
    return 0;
}
```

작은 전용아이콘을 그림 7-6 에 보여 주었다.(물론 전용아이콘을 다른 모양으로 작성해도 상관 없다.) 전용유표는 창문우에 마우스를 가져갔을 때 표시된다.

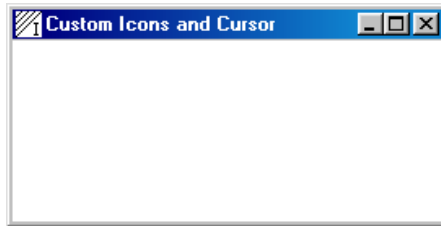



그림 7-6. 작은 전용아이콘

LoadImage( )의 사용방법

앞서 본 실례프로그람들에서는 아이콘과 유표를 적재하는데 *LoadIcon( )* 및 *LoadCursor( )*를 사용하였다. 비트매프의 화상을 적재하는데는 LoadBitmap( )를 사용하였다. 이 함수들은 초기의 Windows 에서부터 사용되여 온것들이다. 이 함수들의 사용에서는 현재도 전혀 문제가 없다. 그러나 Windows 2000 은 이것들을 대신하여 더 편리하게 사용할수 있는 *LoadImage( )*라는 함수를 제공하고 있다.

*LoadImage( )*는 유연한 기능을 가진 함수로서 아이콘, 유표 및 임의의 크기의 비트매프를 적재하는데 사용되며 더우기 그것들을 적재할 때 세밀한 설정을 할수도 있다. 실례로 *LoadImage( )*를 사용하여 큰 아이콘의 파일안에 정의되여 있는것이 아닌 다른 작은 아이콘을 적재할수 있다.

이 책의 실례프로그람들에서는 *LoadImage( )*가 가지는 기능이 필요되지는 않으므로 함수의 기능에 대해서만 설명해 둔다.

*LoadImage( )*의 선언은 다음과 같다.

```
HANDLE LoadImage(HINSTANCE hThisInst, LPCSTR lpszName,
                    UINT what, int width, int height, UINT how);
```

실체의 손잡이는 hThisInst 에 설정한다. 적재할 자원이름의 지시자를 lpszName 에 설정한다. what 의 값으로는 적재할 화상의 종류를 설정한다. 이것은 아래의 어느 한 값이여야 한다.

IMAGE_ICON IMAGE_CURSOR IMAGE_BITMAP

화상의 너비와 높이를 width 와 height 에 설정한다. how 에는 화상의 적재방법을 지시하는 값을 설정하며 다음의 두개의 값이 자주 사용된다.

LR_DEFAULTSIZE LR_LOADFROMFILE

LR_DEFAULTSIZE 가 설정되고 width 와 height 가 령인 경우는 아이콘과 유표의 표준적인 크기가 너비와 높이로서 리용된다. LR_DEFAULTSIZE 가 설정되여 있지 않고 width 와 height 가 령인 경우는 실제 화상의 크기가 그대로 너비와 높이로 리용된다. *LR_LOADFROMFILE* 이 지정된 경우는 lpszName 이 자원파일안에 정의된 자원의 이름이 아니라 화상이 보관된 파일자체의 이름을 가리키는것으로 된다.

LoadImage( )는 호출이 성공하면 화상의 손잡이를 돌려 주며 오유가 발생한 경우에는 NULL 을 돌려 준다.

LoadImage( )의 기능을 시험해 보기 위해 제일 처음 작성한 비트매프의 실례프로그람에서 WM_CREATE 를 처리하는 부분을 아래의 프로그람코드로 바꾸어보자.

```
hbit = (HBITMAP) LoadImage(hInst, “MyBP”, IMAGE_BITMAP, 256, 128, 0);
```

프로그람의 동작은 달라 지지 않는다는것을 알게 될것이다. 이번에는 비트매프의 크기를 변경해 보자. 례를 들어 126×64 로 변경하여 결과를 확인해 보면 비트매프의 일부만이 적재되는것을 보게 된다.

LoadImage( )를 사용하면 프로그람에서 화상을 적재할 때 세밀한 조종을 할수 있으나 이 책의 실례프로그람들에서는 그러한 기능들을 필요로 하지 않으므로 다음 장부터는 LoadImage( )를 사용하지 않는다.

제 8 장

본문의 조종

Windows 2000 은 창문의 의뢰자구역에 본문을 출력하기 위한 세밀한 조종을 할수 있는 세련된 기능들을 제공하고 있다.

앞장들에서는 본문을 출력하는데 가장 기초적인 방법만을 리용하였다.

실례로 행간격이나 문자간격으로는 적당한 값이나 고정된 값을 리용하였다. 물론 이 수법은 사실 정확치 못한 방법이다. 왜냐하면 Windows 2000 에서는 비례적기호서체가 사용되며 서체의 크기를 변경할수 있기때문이다.

이 장에서는 본문의 출력을 조종하기 위한 정확한 방법에 대해 설명한다.

Windows 2000 에서는 다른 많은 요소와 마찬가지로 프로그람작성자들이 본문의 출력에 있어서 그의 크기, 무게 및 서체형 등의 설정을 거의 제한없이 조종할수 있다. 필요한 서체의 종류를 선택하고 그것을 필요에 따라 변경시킬수도 있다. 또한 본문의 기초선의 방향을 변경시켜 수평방향으로만이 아니라 경사 또는 수직방향으로 표시할수도 있다.

물론 본문출력에 대한 고급한 조종을 실현하려면 그만한 노력이 필요하게 된다. 프로그람에서 진행해야 하는 처리량은 상상밖으로 많아 지게 된다. 그러나 얻어지는 결과는 기울인 노력에 충분히 맞먹는것으로 된다.

이 장에서는 먼저 창문의 자리표계와 본문을 넘기기하는 방법에 대해 설명한다. 다음 창문의 의뢰자구역에로의 본문출력을 조종하는데 사용되는 몇가지 API 함수를 설명한다. 그 함수들의 사용방법과 의뢰자구역에 적절한 간격으로 본문을 표시하는 방법을 보여 주는 실례프로그람을 작성한다. 또한 이 장에서는 Windows 가 본문의 서체를 취급하는 방법에 대해서도 설명한다.

창문의 자리표

Window 에는 TextOut( )라는 본문출력함수가 있다. 이 함수는 항상 창문에 대한 상대적인 자리표를 지정하여 문자렬을 표시한다. 그러므로 화면상의 어느 위치에 창문이 있어도 TextOut( )에 보내는 자리표에는 영향이 없다. 체계설정으로 창문의 의뢰자구역의 왼쪽웃모서리가 자리표원점인 (0, 0)으로 된다. X 자리표의 값은 창문의 오른쪽으로 가면서 증가하며 Y 자리표의 값은 창문의 밑으로 내려 가면서 증가한다.

지금까지는 TextOut( )에 지정하는 창문의 자리표를 그것이 실제로 어떻게 정의되는가를 생각하지 않고 사용하였다. 그러므로 여기서 몇가지 개념을 명백히 하자. 우선 TextOut( )에서 사용되는 자리표는 론리자리표이라는것이다. 이것은 TextOut( ) 또는 다음 장에서 설명하는 도형함수들을 비롯한 다른 표시함수들에서 사용되는 단위가 론리단위라는것이다.

Windows 는 출력내용을 실제로 화면에 표시할 때 론리단위를 화소단위로 넘기기한다. 이러한 차이를 구별하지 않아도 문제가 생기지 않은것은 체계설정으로는 론리단위가 화소단위와 같게 되여 있기때문이다. 그러나 이 편리한 체계설정이 아니라 다른 넘기기방식이 사용되는 경우도 있다.

본문과 배경색의 설정

TextOut( )를 사용하여 창문에 본문을 출력하면 체계설정으로는 현재의 배경색의 우에 검은색으로 본문이 표시된다. 그러나 *SetTextColor( )* 및 *SetBkColor( )*라는 API 함수를 사용하면 본문과 배경의 색을 설정할수 있다. 이 함수들은 다음과 같이 선언되였다.

```
COLORREF SetTextColor(HDC hdc, COLORREF color);

COLORREF SetBkColor(HDC hdc, COLORREF color);
```

SetTextColor( )는 hdc 에서 지정된 장치의 본문색을 color 에서 지정된 색(혹은 장치가 표시할수 있는 색중에서 제일 류사한 색)으로 설정한다. SetBkColor( )는 본문의 배경을 color 로 지정된 색(혹은 가장 가까운 색)으로 설정한다. 두 함수는 바로 전에 설정되여 있던 색을 돌림값으로서 돌려 준다. 오유가 발생한 경우는 *CLR_INVALID*라는 값을 돌려 준다.

색은 *COLORREF 형*으로 지정한다. 이것은 32bit 의 옹근수값이다. Windows 2000 에서 색을 지정하는데는 세가지 방법이 있다.

첫번째 방법은 RGB(적, 록, 청)값을 사용하는것이다. 이것은 가장 일반적인 방법이다. RGB 의 값은 세가지 색의 농도가 혼합된 색을 지정한다. 두번째 방법은 론리조색판의 색인을 사용하는것이다. 세번째 방법은 조색판의 *RGB 값*을 사용하는것이다. 이 장에서는 첫번째 방법에 대해서만 설명하기로 한다.

RGB 색을 보관하기 위한 긴 정수값(COLORREF)은 아래와 같이 부호화되여 그것이 SetTextColor( ) 및 SetBkColor( )에 전달된다.

byte	색
byte 0(제일 아래 byte)	붉은색
byte 1	푸른색
byte 2	풀색
byte 3(제일 웃 byte)	령이여야 한다.

RGB 의 세 가지 색은 각각 0~255 의 범위에서 값을 설정할수 있으며 0 은 가장 희박한 농도를, 255 는 가장 짙은 농도를 가리킨다.

직접 COLORREF 의 값을 작성해도 상관이 없지만 Windows 는 그 작업을 자동화하여 주는 *RGB( )*마크로를 정의하고 있다. 다음에 서식을 보여 주었다.

COLORREF RGB(BYTE red, BYTE green, BYTE blue);

red, green 및 blue 의 값은 0~255 의 범위에 속해야 한다. 그러므로 밝은 분홍색을 작성하려면 RGB(255, 0, 255)를 사용하면 된다. 흰색을 작성한다면 RGB(255, 255, 255)를 사용한다. 검은색을 작성한다면 RGB(0, 0, 0)을 사용한다. 그밖의 색을 작성한다면 세가지 기본색의 농도를 설정하고 그것들을 조합한다. 실례로 RGB(0, 100, 100)은 밝은 물색을 작성한다. 실제로 시험해 보고 어떤 색이 응용프로그람에 적당한가를 결정하시오.

배경현시방식의 설정

*SetBkMode( )*라는 API 함수를 사용하면 화면에 본문이 표시되는 때의 배경방식을 조종할수 있다. 선언은 다음과 같다.

int SetBkMode(HDC hdc, int mode);

이 함수는 본문 또는 다른 어떤 출력이 표시될 때 배경을 어떻게 표시하는가를 지정

한다. 장치상황의 손잡이를 hdc에 설정한다. 배경방식을 mode에 설정한다. mode의 값으로는 *OPAQUE* 또는 *TRANSPARENT* 중에서 어느 하나를 설정한다. 이 함수는 바로 전에 설정되여 있던 값을 돌려 주며 오유가 발생한 경우에는 령을 돌려 준다.

mode 가 OPAQUE 인 경우에 본문이 출력되면 배경이 현재의 배경색으로 변한다. mode 가 TRANSPARENT 인 경우에는 배경이 변하지 않는다. 이 경우에는 SetBkColor( )로 설정한 배경색이 무시된다. 체계설정의 배경방식은 OPAQUE 이다.

본문치수의 얻기

Windows 에서는 모든 본문의 *서체*가 비례적인것으로 되여 있으며 문자의 크기도 다르다. 실례로 "i"라는 문자는 "w"라는 문자와 너비가 다르다. 매 문자의 높이와 디센더(기초선아래의 부분)의 길이도 서체에 따라 다르다. 기초선의 간격도 변경할수 있게 되여 있다.

Windows 는 창문의 의뢰자구역에 본문을 출력하는 기능을 가진 함수를 몇종류밖에 제공하지 않고 있다. 주요한 함수는 지금까지 사용해온 TextOut( )함수이다. 이 함수에는 한가지 기능밖에 없다. 지정된 위치에 문자렬을 표시하는것이다. 례하면 출력서식을 설정하거나 복귀 및 개행을 자동적으로 처리하는 기능 등은 가지고 있지 않다. 그러므로 필요한 처리는 모두 프로그람작성자들이 해야 한다.

매 문자의 크기가 다르다는것과 프로그람의 실행중에 서체의 종류를 변경할수 있다는 점으로부터 현재 선택되여 있는 서체의 크기나 다른 속성을 알아내는 방법이 필요하게 된다. 실례로 행을 바꾸어 문자렬을 출력하려면 서체의 높이와 행간의 화소수를 알기 위한 어떠한 방법이 필요하게 된다.

현재의 서체정보를 얻는 API 함수는 *GetTextMetrics( )*이다. 아래에 선언을 보여 주었다.

```
BOOL GetTextMetrics(HDC hdc, LPTEXTMETRIC lpTAttrib);
```

hdc 는 출력장치의 손잡이이며 일반적으로 GetDC( )나 BeginPaint( )로 얻는 다. lpAttrib 는 TEXTMETRIC 구조체의 지시자이다. 이 구조체에는 현재 선택되여 있는 본문치수가 돌려 진다. 아래에 *TEXTMETRIC* 구조체의 정의를 보여 주었다.

```
typedef struct tagTEXTMETRIC
{
        LONG tmHeight;             // 서체의 전체 높이
        LONG tmAscent;             // 기초선의 높이
        LONG tmDescent;            // 디센더의 길이
        LONG tmInternalLeading;    // 기호의 웃부분의 간격
```

```
            LONG tmExternalLeading;          // 행간의 빈 간격
            LONG tmAveCharWidth              // 평균너비
            LONG tmMaxCharWidth              // 최대너비
            LONG tmWeight;                   // 무게
            LONG tmOverhang;                 // 특수한 서체에 추가되는 너비
            LONG tmDigitizedAspectX;         // 수평 비률
            LONG tmDigitizedAspectY;         // 수직 비률
            LONG tmFirstChar;                // 서체의 선두 문자
            LONG tmLastChar;                 // 서체의 마지막 문자
            LONG tmDefaultChar;              // 체계설정의 문자
            LONG tmBreakChar;                // 단어를 끝내는 문자
            LONG tmItalic;                   // 경사체이면 령 아닌 값
            LONG tmUnderlined;               // 밑선이 있으면 령 아닌 값
            LONG tmStruckOut;                // 취소선이 있으면 령 아닌 값
            LONG tmPitchAndFamily;           // 서체의 간격과 계렬
            LONG tmCharSet;                  // 문자모임의 식별자
}   TEXTMETRIC;
```

다시 한보 전진

NEWTEXTMETRIC 와 NEWTEXTMETRICEX

TEXTMETRIC 를 확장한 NEWTEXTMETRIC 라는 구조체가 있다. *NEWTEXTMETRICEX*는 TEXTMETRIC의 마감에 4개의 성원을 추가한것이다. 추가된 성원들은 TrueType*서체*를 지원하기 위한것들이다.(TrueType서체는 우수한 확대축소(scalibility)기능을 제공한다.) NEWTEXTMETRIC 에 추가된 성원들은 다음과 같다.

```
DWORD ntmFlags;              // 서체의 형식을 가리킨다.
UINT ntmSizeEM;              // “M”이라는 기호의 크기
UINT ntmCellHeight;          // 서체의 높이
UINT ntmAvgWidth;            // 평균너비
```

NEWTEXTMETRIC 의 확장판으로서 최근 Win32 에 추가된 구조체에 NEWTEXTMETRICEX 라는것도 있다. 정의는 다음과 같다.

```
typedef struct tagNEWTEXTMETRICEX
{
```

```
        NEWTEXTMETRIC ntmTm;
        FONTSIGNATURE ntmFontSig;     // 서체서명
    } NEWTEXTMETRICEX;
```

NEWTEXTMETRICEX 는 NEWTEXTMETRIC 에 FONTSIGNATURE 구조체를 추가한것이다. FONTSIGNATURE 구조체는 유니코드와 코드페지에 대한 정보를 보관한다.

이 장에서는 NEWTEXTMETRIC 나 NEWTEXTMETRICEX 에 추가된 성원들을 필요로 하지 않는다. 그러나 실제적인 프로그람들에서는 이것들이 가치있는것으로 될것이다.

GetTextMetrics( )에서 얻은 대부분의 값들은 이 장에서 쓰이지 않지만 본문의 기초선의 간격을 계산하는데 필요되는 두개의 값은 매우 중요하다. 행간 간격은 창문에 한 행이상의 본문을 출력하려는 경우에 필요하게 된다.

서체가 한가지뿐이고 그 크기가 고정되여 있는 조종탁(Console)프로그람과는 달리 Windows 에는 여러가지 종류의 서체가 있으며 그의 크기를 변경시킬수도 있다. 매개 서체는 문자의 높이와 행간격을 정의하고 있다. 그러므로 사전에 본문행사이의 적절한 수직거리(Y)를 아는것은 불가능하다.

본문의 다음 행의 표시위치를 결정하려면 GetTextMetrics( )를 호출하여 tmHeight 및 tmExternalLeading 의 두개의 값을 얻어야 한다. 이 값들에는 문자의 높이 및 행사이에 배치해야 할 공백간격이 보관된다. 이 두개의 값을 더하여 본문의 행간격을 구할수 있다.

tmExternalLeading에는 본문의 행사이에 놓이는 공백값만이 보관되여 있다는데 주의해야 한다. 이 값은 서체자체의 높이와는 다르다. 행의 전체 높이를 계산하려면 항상 두 값이 필요하게 된다.

문자렬의 길이를 구하기

Windows 는 본문유표를 자동적으로 조종하거나 현재의 표시위치를 관리하는 일은 해 주지 않는다. 그러므로 같은 행에서 문자를 련속 표시하려면 바로 전의 출력에서 마감위치를 기록해 둘 필요가 있다. 그러므로 문자렬의 길이를 론리단위로 알기 위한 어떤 방법이 필요하게 된다.

대부분의 서체에서는 매개 문자가 같은 크기로 되여 있지 않으므로 단순히 문자수만 알고서는 문자렬의 길이를 론리단위로 알수 없다. 다시말하여 매 문자의 너비가 다르므로 창문의 표시위치를 결정하는데 *strlen( )*를 사용할수 없다.

이 문제를 해결하기 위해 Windows 2000 은 *GetTextExtentPoint32( )*라는 API 함수를 제공하고 있다. 아래에 선언을 보여 주었다.

```
BOOL GetTextExtentPoint32(HDC hdc, LPCSTR lpszString, int len,
                          LPSIZE lpSize);
```

hdc는 출력장치의 손잡이이다. 길이를 알려는 문자렬을 lpszString에 설정한다. 문자렬의 길이를 len 에 설정한다. 문자렬의 너비와 높이는 론리단위로 돌려 지며 lpSize가 가리키는 SIZE구조체에 보관된다. 아래에 *SIZE*구조체의 정의를 보여 주었다.

```
typedef struct tagSIZE {
    LONG cx;      // 너비
    LONG cy;      // 높이
} SIZE;
```

GetTextExtentPoint32()의 돌림값의 cx성원에는 문자렬의 너비가 보관되여 있다. 표시된 문자렬의 마감위치를 판정하기 위해 이 값을 사용할수 있다. 같은 행에 문자렬을 표시하는 경우에는 바로 전에 출력된 문자의 마감위치에서부터 계속해 나가면 된다. GetTextExtentPoint32( )는 호출이 성공하면 령 아닌 값을 돌려 주며 실패하면 령을 돌려 준다.

이식과 관련한 요점 : GetTextExtentPoint32( )는 Windows 3.1 프로그람에서 사용되고 있는 낡은 16bit 함수인 GetTextExtent( )를 대신하는 함수이다.

본문의 간단한 실례

실례 8-1 에 보여 준 프로그람은 본문의 출력과 지금까지 설명한 본문과 관련된 함수들의 실례를 보여 주는 프로그람이다. WM_PAINT 통보문을 받으면 기본창문의 의뢰자구역에 여러 행의 본문이 표시된다.

실례 8-1. Text 프로그람

```
// 본문출력의 관리


#include <windows.h>
#include <cstring>
#include <cstdio>
```

```
LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);

char szWinName[] = "MyWin"; // 창문클라스의 이름

char str[255]; // 출력할 문자렬을 보관한다.
int X=0, Y=0;  // 현재의 출력위치

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;

  // 창문클라스를 정의한다.
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst;       // 실체의 손잡이
  wcl.lpszClassName = szWinName; // 창문클라스의 이름
  wcl.lpfnWndProc = WindowFunc;  // 창문함수
  wcl.style = 0;                   // 체계설정의 형식

  wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
  wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
  wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

  wcl.lpszMenuName = NULL; // 클라스차림표는 없다.
  wcl.cbClsExtra = 0;       // 보조기억기령역은 필요 없다.
  wcl.cbWndExtra = 0;

  // 창문의 배경색을 흰색으로 한다.
  wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

  // 창문클라스를 등록한다.
  if(!RegisterClassEx(&wcl)) return 0;

  /* 창문클라스가 등록되였으므로
```

```
     창문을 작성할수 있다. */
  hwnd = CreateWindow(
    szWinName, // 창문클라스의 이름
    "Managing Text Output", // 제목
    WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
    CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
    CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
    CW_USEDEFAULT, // 너비는 Windows가 결정하게 한다.
    CW_USEDEFAULT, // 높이는 Windows가 결정하게 한다.
    NULL,          // 어미창문은 없다.
    NULL,          // 차림표는 없다.
    hThisInst,     // 실체의 손잡이
    NULL           // 추가파라메터는 없다.
  );

  // 창문을 표시한다.
  ShowWindow(hwnd, nWinMode);
  UpdateWindow(hwnd);

  // 통보문순환고리를 작성한다.
  while(GetMessage(&msg, NULL, 0, 0))
  {
    TranslateMessage(&msg); // 건반통보를 변환한다.
    DispatchMessage(&msg);  // Windows 2000에 조종을 넘긴다.
  }
  return msg.wParam;
}

/* 이 함수는 Windows 2000으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을  받아 들인다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  HDC hdc;
  TEXTMETRIC tm;
  SIZE size;
  PAINTSTRUCT paintstruct;
```

```
switch(message) {
  case WM_PAINT:
    hdc = BeginPaint(hwnd, &paintstruct);

     // 본문치수를 얻는다.
    GetTextMetrics(hdc, &tm);

    X = Y = 0;

    sprintf(str, "This is on the first line.");
    TextOut(hdc, X, Y, str, strlen(str));
    Y = Y + tm.tmHeight + tm.tmExternalLeading; // 행바꾸기한다.

    strcpy(str, "This is on the second line. ");
    TextOut(hdc, X, Y, str, strlen(str));
    Y = Y + tm.tmHeight + tm.tmExternalLeading; // 행바꾸기한다.

    strcpy(str, "This is on the third line. ");
    TextOut(hdc, X, Y, str, strlen(str));

    // 문자렬의 길이를 구한다.
    GetTextExtentPoint32(hdc, str, strlen(str), &size);
    sprintf(str, "The preceding sentence is %ld units long",
            size.cx);
    X = size.cx; // 바로 전의 문자렬의 마감위치로 간다.
    TextOut(hdc, X, Y, str, strlen(str));
    Y = Y + tm.tmHeight + tm.tmExternalLeading; // 개행한다.
    X = 0; // 행의 선두위치로 복귀한다.

    sprintf(str, "The space between lines is %ld pixels.",
            tm.tmExternalLeading+tm.tmHeight);
    TextOut(hdc, X, Y, str, strlen(str));
    Y = Y + tm.tmHeight + tm.tmExternalLeading; // 개행한다.

    sprintf(str, "Average character width is %ld pixels",
            tm.tmAveCharWidth);
    TextOut(hdc, X, Y, str, strlen(str));
    Y = Y + tm.tmHeight + tm.tmExternalLeading; // 개행한다.

    // 본문색을 붉은색으로 한다.
    SetTextColor(hdc, RGB(255, 0, 0));
```

```
        // 배경색을 푸른 색으로 한다.
        SetBkColor(hdc, RGB(0, 0, 255));

        sprintf(str, "This line is red on blue background.");
        TextOut(hdc, X, Y, str, strlen(str));
        Y = Y + tm.tmHeight + tm.tmExternalLeading; // 개행한다.

        // TRANSPARENT 방식으로 변경한다.
        SetBkMode(hdc, TRANSPARENT);
        sprintf(str,
          "This line is red. The background is unchanged.");
        TextOut(hdc, X, Y, str, strlen(str));

        EndPaint(hwnd, &paintstruct);
        break;
      case WM_DESTROY: // 프로그람을 끝낸다.
        PostQuitMessage(0);
        break;
      default:
        /* 이 switch 문에서 지정된것 이외의 통보문은
           Windows 2000 에 처리를 맡긴다. */
       return DefWindowProc(hwnd, message, wParam, lParam);
    }

    return 0;
  }
```

이 프로그람의 실행결과를 그림 8-1 에 보여 주었다.

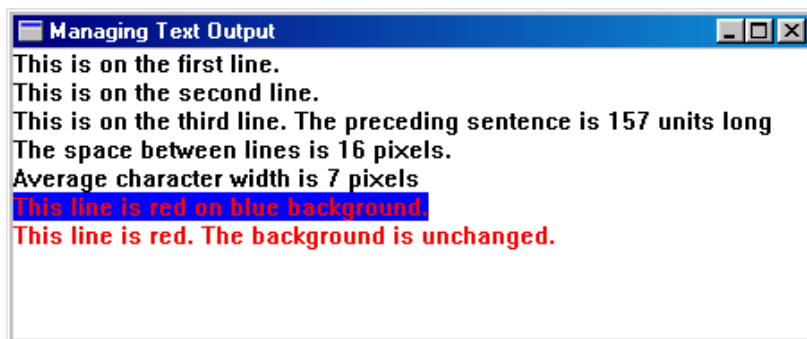



그림 8-1. 본문을 출력하는 실례프로그람의 실행결과

그러면 프로그람의 내용을 자세히 설명해 보자. 우선 두개의 대역변수 X 와 Y 가 선언되고 그것들이 령으로 초기화되고 있다. 이 변수들은 본문의 현재표시위치를 보관한다. 변수값은 출력이 진행될 때마다 갱신된다.

이 프로그람에서 중요한 부분은 대체로 WM_PAINT 통보문안에 있다. WM_PAINT 통보문을 받으면 장치상황이 얻어지고 또한 본문치수가 얻어 진다. 다음 제일 첫행의 본문이 출력된다. 이 본문은 *sprintf( )*를 사용하여 작성되며 *TextOut( )*를 사용하여 출력된다.

TextOut( )에는 본문의 서식을 설정하는 기능이 없으므로 미리 출력내용을 작성해 두어야 한다. 문자렬이 표시되면 서체의 높이와 행간격을 더하여 Y 자리표를 다음 행으로 옮긴다.

계속하여 프로그람은 “This is on the second line.” 및 “This is on the third line.” 라는 문자렬을 한 행씩 출력한다. 이때 세번째 행의 문자렬의 길이가 GetTextExtent 32( )를 사용하여 얻어 진다. 이 값은 다음의 문자렬을 표시하기전에 바로 전의 문자렬의 마감위치에 X 자리표를 옮기는데 사용된다.

여기서는 Y 자리표를 변경시키지 않으므로 다음 문자렬이 마지막으로 표시된 문자렬의 끝에 련결되여 표시되며 개행이 진행되지 않는다. 다음 문자렬을 표시하기전에 프로그람은 Y 자리표를 다음 행으로 옮기고 X 자리표를 왼쪽 한계자리표인 령으로 재설정한다. 이렇게 되여 제일 첫행의 출력과 마찬가지로 복귀와 개행이 진행된다.

다음 현재 선택되여 있는 서체와 관련된 몇가지 정보가 표시된다. 또한 배경색을 청색으로 설정하고 본문을 적색으로 설정하고 나서 문자렬을 표시한다. 마지막으로 배경방식을 TRANSPARENT 로 설정한 다음 문자렬을 표시한다. 이 경우에는 청색의 배경색이 무시된다.

서체의 조종

Windows 2000 및 Windows 전반에서는 사용자대면부의 거의 모든 부분을 프로그람에서 자유로이 조종할수 있게 되여 있다. 고도로 풍부한 본문조종기능을 제공하고 있는 것도 그의 한 실례이다.

본문조종기능의 하나로서 여러가지 종류의 서체가 제공되고 있다. 응용프로그람에서 사용되는 서체를 조종하여 다른 프로그람과 구별되는 독특한 외형을 가진 응용프로그람을 작성할수 있다.

Windows 2000 에는 몇개의 *내장서체*가 있으며 *전용서체*를 작성할수도 있다. 여기에서는 서체의 조종기능을 설명한다. 우선 몇가지 용어의 의미부터 설명해 보자.

서체, 서체계렬, 서체형 및 형식

일반적으로 *서체*란 *문자모임* 및 그것들이 가지는 외형을 의미한다. 일반적으로 사용되는 서체에는 여러가지 종류가 있다. 실례로 Courier 나 Times Roman 등은 가장 대중적인 서체이다.

서체에는 4 가지 속성이 있다. 문자모임, *서체형*, *형식* 및 *크기*이다. 여기서 그것들의 의미에 대해 설명하기로 하자.

Windows 2000 은 여러가지 문자모임을 제공하고 있다. 보통 사용되는것은 Windows 의 표준문자모임이다. 이 문자모임은 ANSI 문자모임에 기초하고 있으므로 매 기호는 8bit 로 구성되며 서유럽의 많은 언어들에 활용할수 있다.

이밖에도 Windows 가 제공하고 있는 문자모임에는 Unicode, OEM(Original Equipment Manufacturer) 및 특수기호(수식의 표기에 사용되는 기호문자의 모임)가 있다.

전용의 문자모임도 사용할수 있다.

서체형(Type face)은 서체의 문자형태와 도안을 정의한다. 다시말하여 서체형은 서체의 고유하고 시각적인 특징을 결정한다.

일반적으로 서체의 형식이란 서체가 표준체, 굵은체 혹은 경사체중 어느것으로 표시되는가를 지정하는것이다. Windows 에서는 서체의 굵기를 아주 세밀하게 조종할수 있으므로 서체의 굵기를 계단식으로 설정할수 있다.

서체의 크기는 1/72 inch 를 의미하는 *포인트*단위로 지정된다. Windows 는 형태에 따라 서체를 다섯개의 *계렬*로 분류하고 있다. 서체계렬의 종류에는 Decorative, Modern, Roman, Script 및 Swiss 가 있다.

Decorative 계렬의 서체는 특수한 서체이다. Modern 계렬의 서체는 비례서체가 아니다. Roman 계렬의 서체는 *비례서체*이며 *받침선*(Serif)을 가지고 있다.(받침선이란 문자의 끝에 붙어 있는 작은 장식선이다.) Script 계렬의 서체는 손으로 쓴 글자같은 형태를 취하고 있다. Swiss 계렬의 서체는 비례서체이며 받침선이 없다.

비례서체에서는 매개 문자의 너비와 간격이 가변형으로 되여 있다. 비례서체를 *가변간격서체*라고도 부른다. 비례서체가 아닌 서체에서는 매개 문자의 너비가 같다. 이것들을 *고정간격서체* 또는 고정령역(Monoscope)서체라고도 부른다.

주사선, 벡토르 및 TrueType 서체

Windows 2000 은 서체를 보관하고 표시하기 위한 4 가지 방법을 제공하고 있다. 그것들은 각각 *주사선*, 벡토르, TrueType 및 OpenType 라고 부른다.

주사선서체는 서체내에 매 *문자획*(Glyph)을 비트매프로서 보관하고 있다. 표시방법은 간단하지만 확대, 축소를 원만하게 할수 없다. 주사선서체는 현시장치나 인쇄기에 출력할 때 가장 잘 사용된다.

*벡토르*서체는 매개 문자획을 형성하는 선분정보를 보관한것이다. 문자를 표시할 때는 이 선분들이 그려진다. 작도기를 사용하는 경우에는 벡토르서체가 가장 리상적이다.

TrueType 서체는 확대,축소를 원활하게 할수 있는 서체이다. 현시장치의 표시내용을 인쇄기출력으로 정확하게 변환할수도 있다. 이런 리유로부터 TrueType 서체는 탁상출판 분야의 응용프로그람에서 잘 사용되고 있다. OpenType 서체는 PostScript 형식의 문자획의 정의를 지원하는 TrueType 서체의 일종이다.

내장서체의 사용방법

Windows 2000 의 *내장서체*는 *GetStockObject( )*를 리용하여 선택하는 내장객체이다. Windows 2000 은 7 개의 내장서체를 제공하고 있다.

아래에 서체의 종류를 가리키는 마크로들을 표시하였다.

서　　체	설　　명
ANSI_FIXED_FONT	고정간격서체
ANSI_VAR_FONT	가변간격서체
DEVICE_DEFAULT_FONT	체계설정의 장치서체
DEFAULT_GUI_FONT	체계설정의 GUI 서체
OEM_FIXED_FONT	OEM 서체
SYSTEM_FONT	Windows 2000 이 사용하는 체계서체
SYSTEM_FIXED_FONT	낡은 판본의 Windows 가 사용하는 체계서체

*체계서체*는 Windows 2000 이 차림표나 대화칸에서 사용하는 서체이다. 낡은 판본의 Windows 에서는 고정간격체계의 서체를 사용하였으나 Windows 3.0 부터 가변간격의 체계의 서체가 사용되게 되였다. Windows 2000 에서도 가변간격체계의 서체가 사용되고 있다.

내장서체를 선택하여 사용하는 방법은 간단하다. 우선 HFONT 형의 서체손잡이를 선언한 다음 서체의 손잡이를 돌려 주는 API 함수인 *GetStockObject()*를 리용하여 목적하는 서체를 적재한다.

서체를 변경하자면 파라메터에 새로운 서체를 지정한 *SelectObject()*를 호출하여 서체를 선택한다. SelectObject( )는 바로 전에 선택되여 있었던 서체의 손잡이를 돌려 주므로 그 값을 보관해 두면 후에 본래의 서체로 복귀할수도 있다.

실례 8-2 의 프로그람코드는 서체를 변경하는 실례이다. 이 프로그람에서는 다시그리기를 진행하는데 제 7 장에서 설명한 *가상창문*기술을 사용하고 있다.

실례 8-2. BuiltinFont 프로그람

```
// 내장서체의 실례
```

```
#include <windows.h>
#include <cstring>
#include <cstdio>
#include "text.h"

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);

char szWinName[] = "MyWin"; // 창문클라스의 이름

char str[255]; // 출력할 문자렬을 보관한다.

int X=0, Y=0;    // 현재의 출력위치
int maxX, maxY; // 화면의 크기

HDC memdc;            // 가상장치의 손잡이를 보관한다.
HBITMAP hbit;         // 가상비트매프를 보관한다.
HBRUSH hbrush;        // 붓의 손잡이를 보관한다.
HFONT holdf, hnewf; // 서체의 손잡이를 보관한다.

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;
  HACCEL hAccel;

  // 창문클라스를 정의한다.
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst;      // 실체의 손잡이
  wcl.lpszClassName = szWinName; // 창문클라스의 이름
  wcl.lpfnWndProc = WindowFunc;  // 창문함수
  wcl.style = 0;                  // 체계설정의 형식

  wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
  wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
  wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식
```

```
wcl.lpszMenuName = "FontMenu"; // 기본차림표

wcl.cbClsExtra = 0;       // 보조기억기령역은 필요 없다.
wcl.cbWndExtra = 0;

// 창문의 배경색을 흰색으로 한다.
wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

// 창문클라스를 등록한다.
if(!RegisterClassEx(&wcl)) return 0;

/* 창문클라스가 등록되였으므로
   창문을 작성할수 있다. */
hwnd = CreateWindow(
  szWinName, // 창문클라스의 이름
  "Using Built-in Fonts", // 제목
  WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
  CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 너비는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 높이는 Windows가 결정하게 한다.
  NULL,          // 어미창문은 없다.
  NULL,          // 차림표는 없다.
  hThisInst,     // 실체의 손잡이
  NULL           // 추가파라메터는 없다.
);

// 건반가속기를 적재한다.
hAccel = LoadAccelerators(hThisInst, "FontMenu");

// 창문을 표시한다.
ShowWindow(hwnd, nWinMode);
UpdateWindow(hwnd);

// 통보문순환고리를 작성한다.
while(GetMessage(&msg, NULL, 0, 0))
{
```

```
      if(!TranslateAccelerator(hwnd, hAccel, &msg)) {
        TranslateMessage(&msg); // 건반통보를 변환한다.
        DispatchMessage(&msg); // Windows 2000에 조종을 넘긴다.
      }
    }

    return msg.wParam;
  }

  /* 이 함수는 Windows 2000으로부터 호출되여
     통보문대기렬에서 꺼낸 통보문을 받아 들인다.
  */
  LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                              WPARAM wParam, LPARAM lParam)
  {
    HDC hdc;
    PAINTSTRUCT paintstruct;
    static TEXTMETRIC tm;
    SIZE size;
    static fontswitch = 0;
    int response;

    switch(message) {
      case WM_CREATE:
        // 화면의 크기를 얻는다.
        maxX = GetSystemMetrics(SM_CXSCREEN);
        maxY = GetSystemMetrics(SM_CYSCREEN);

        // 가상창문을 작성한다.
        hdc = GetDC(hwnd);
        memdc = CreateCompatibleDC(hdc);
        hbit = CreateCompatibleBitmap(hdc, maxX, maxY);
        SelectObject(memdc, hbit);
        hbrush = (HBRUSH) GetStockObject(WHITE_BRUSH);
        SelectObject(memdc, hbrush);
        PatBlt(memdc, 0, 0, maxX, maxY, PATCOPY);
        // 새로운 서체를 얻는다.
        hnewf = (HFONT) GetStockObject(ANSI_VAR_FONT);
```

```
      ReleaseDC(hwnd, hdc);
      break;
    case WM_COMMAND:
      switch(LOWORD(wParam)) {
        case IDM_SHOW:
          // 본문색을 검은색으로, 배경방식은 TRANSAPARENT 로 설정한다.
          SetTextColor(memdc, RGB(0, 0, 0));
          SetBkMode(memdc, TRANSPARENT);

          // 본문치수를 얻는다.
          GetTextMetrics(memdc, &tm);

          sprintf(str, "The font is %ld pixels high.",
                  tm.tmHeight);
          TextOut(memdc, X, Y, str, strlen(str));
          Y = Y + tm.tmHeight + tm.tmExternalLeading; // 개행한다.

          strcpy(str, "This is on the next line. ");
          TextOut(memdc, X, Y, str, strlen(str));

          // 문자렬의 길이를 구한다.
          GetTextExtentPoint32(memdc, str, strlen(str), &size);
          sprintf(str, "Previous string is %ld units long",
                  size.cx);
          X = size.cx; // 전 문자렬의 마감위치로 이동한다.
          TextOut(memdc, X, Y, str, strlen(str));
          Y = Y + tm.tmHeight + tm.tmExternalLeading; // 개행한다.
          X = 0; // X 자리표를 재설정한다.

          sprintf(str, "Screen dimensions: %d %d", maxX, maxY);
          TextOut(memdc, X, Y, str, strlen(str));
          Y = Y + tm.tmHeight + tm.tmExternalLeading; // 개행한다.
          InvalidateRect(hwnd, NULL, 1);
          break;
        case IDM_RESET:
          X = Y = 0;
```

```
        // 배경을 색칠하여 지운다.
        PatBlt(memdc, 0, 0, maxX, maxY, PATCOPY);

        InvalidateRect(hwnd, NULL, 1);
        break;
      case IDM_FONT:
        if(!fontswitch) {  // 새로운 서체로 반전절환한다.
          holdf = (HFONT) SelectObject(memdc, hnewf);
          fontswitch = 1;
        }
        else { // 본래의 서체로 반전절환한다.
          SelectObject(memdc, holdf);
          fontswitch = 0;
        }
        break;
      case IDM_EXIT:
        response = MessageBox(hwnd, "Quit the Program?",
                              "Exit", MB_YESNO);
        if(response == IDYES) PostQuitMessage(0);
        break;
      case IDM_HELP:
        MessageBox(hwnd,
                   "F2: Display\nF3: Change Font\nF4: Reset",
                   "Font Fun", MB_OK);
        break;
    }
    break;
  case WM_PAINT: // 다시그리기요구를 처리한다.
    hdc = BeginPaint(hwnd, &paintstruct); // 장치상황을 얻는다.

    // 가상창문을 화면에 복사한다.
    BitBlt(hdc, 0, 0, maxX, maxY, memdc, 0, 0, SRCCOPY);

    EndPaint(hwnd, &paintstruct); // 장치상황을 해제한다.
```

```
        break;
      case WM_DESTROY: // 프로그람을 끝낸다.
        DeleteDC(memdc);
        PostQuitMessage(0);
        break;
      default:
        /* 이 switch 문에서 지정된것 이외의 통보문은
           Windows 2000 에 처리를 맡긴다. */
        return DefWindowProc(hwnd, message, wParam, lParam);
    }

    return 0;
}
```

이 프로그람에서 사용되는 자원파일을 아래에 보여 주었다.

```
#include <windows.h>
#include "text.h"

FontMenu MENU
{
  POPUP "&Fonts" {
    MENUITEM "&Display\tF2", IDM_SHOW
    MENUITEM "Change &Font\tF3", IDM_FONT
    MENUITEM "&Reset\tF4", IDM_RESET
    MENUITEM "E&xit\tCtrl+X", IDM_EXIT
  }
  MENUITEM "&Help", IDM_HELP
}

FontMenu ACCELERATORS
{
  VK_F1, IDM_HELP, VIRTKEY
  VK_F2, IDM_SHOW, VIRTKEY
```

```
    VK_F3, IDM_FONT, VIRTKEY
    VK_F4, IDM_RESET, VIRTKEY
    "^X", IDM_EXIT
}
```

머리부파일 TEXT.H의 내용을 아래에 보여 주었다.

```
#define IDM_SHOW     100
#define IDM_FONT     101
#define IDM_RESET    102
#define IDM_EXIT     103
#define IDM_HELP     104
```

이 프로그람의 실행결과를 다음의 그림 8-2에 보여 주었다.

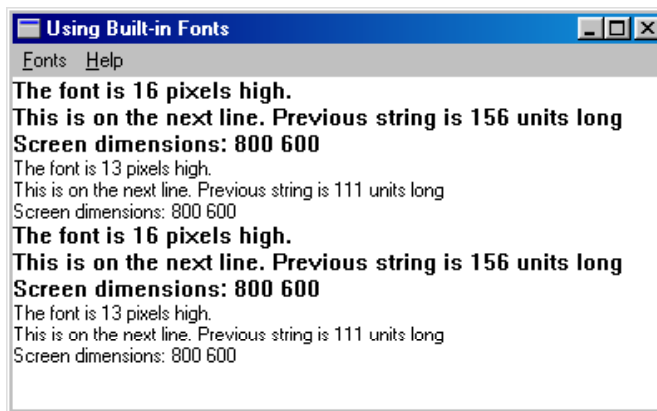



그림 8-2. 내장서체프로그람의 실행결과

이 프로그람의 동작은 다음과 같다. [Display]차림표를 선택하면 현재 선택되여 있는 서체로서 본문이 출력된다. [Change Font]차림표를 선택하면 ANSI 가변간격서체와 체계설정서체가 서로 바뀌면서 선택된다. [Reset]차림표를 선택하면 창문의 표시내용을 소거할수 있다. 이 경우에는 X 자리표와 Y 자리표가 령으로 재설정되고 PATCOPY 를 지정한 *PatBlt( )*를 호출하여 가상창문의 배경이 색칠된다.

다시 한보 전진

Unicode

영어나 도이췰란드어 등 서유럽나라들의 언어에서 사용되는 자모들은 *ANSI 문자모임*을 사용하여 표시할수 있다. ANSI 문자모임은 8bit 문자를 사용하

므로 256종류이상의 문자를 표시할수 없다.(ANSI문자모임은 7bit의 ASCII코드의 확장모임이다.)

이것은 조선어나 중어처럼 256종류이상의 문자를 필요로 하는 일부 아시아나라들의 언어에서 문제로 된다. 이러한 언어에도 대응하기 위해 Unicode라는 *문자모임*이 작성되였다. Unicode에서는 한 문자를 16bit로 표시하므로 매우 큰 문자모임을 실현할수 있다.(16536문자까지) 호환성을 유지하기 위해 Unicode의 선두 256문자는 ANSI문자모임에서 정의된 문자들과 일치하도록 되여 있다.

Windows 2000은 프로그람의 국제대응판을 쉽게 작성할수 있도록 Unicode를 전면적으로 지원하고 있다. Unicode와 코드변환을 진행하는 기능을 가진 Win32 API함수들이 많이 제공되고 있다. 실례로 ToUnicode( )는 가상건코드를 Unicode로 변환하는 API이다. Windows 2000은 프로그람의 번역시의 설정에 따라 Unicode 또는 ASCII 코드의 어느 하나로 자동적으로 넘기기되는 범용적인 자료형도 제공하고 있다.

전용서체의 작성

*전용서체*를 작성하는것을 상당히 복잡한것으로 생각할수도 있으나 실제로는 아주 간단하다. 전용서체를 사용하면 두가지 우점이 있다. 첫번째 우점은 응용프로그람에 고유한 외적특징을 주는것이다. 다음으로 자체의 서체(전용서체)를 작성하여 본문의 출력을 정확히 조종할수 있다는것이다.

전용서체를 작성할 때는 *서체형*을 정의할 필요가 없다. 그대신에 기존의 서체형에 변경을 가하여 목적하는 도안으로 하면 된다. 그러므로 작성하는 서체의 매개 문자들의 도안을 정의할 필요는 없다.

전용서체를 작성하려면 *CreateFont( )*라는 API 함수를 사용한다. 아래에 선언을 보여 주었다.

```
HFONT CreateFont(int Height, int Width, int Escapement,
                 int Orientation, int Weight,
                 DWORD Ital, DWORD Underline,
                 DWORD StrikeThru, DWORD Charset,
                 DWORD Precision, DWORD ClipPrecision,
                 DWORD Quality, DWORD PitchFam,
                 LPCSTR TypefaceName);
```

서체의 높이를 Height 에 설정한다. Height 가 령인 경우에는 체계설정의 높이가 사용된다. 서체의 너비를 Width 에 설정한다. Width 가 령인 경우에는 현재의 확대축소비에 따라 Windows 가 적당한 값을 설정해 준다. Height 와 Width 는 다 론리단위로 설정한다.

창문에 대하여 본문을 임의의 각도로 표시할수도 있다. 이 각도는 Escapement 에 설정한다. 일반적인 수평방향의 본문에서는 이 값을 령으로 한다. 본문의 각도를 설정하는 경우에는 시계바늘회전방향의 반대로 0.1°를 단위로 하여 값을 지정한다. 실례로 값을 900 으로 설정하면 본문이 90° 회전하므로 수직방향으로 출력되게 된다.

문자의 경사도도 Orientation 을 리용하여 설정할수 있다. 여기에서도 수평선에 대해 시계바늘회전방향의 반대로 0.1°를 단위로 값을 설정한다.

참고 *: 체계설정으로는 CreateFont( )의 파라메터인 Escapement 와 Orientation 에 같은 값을 설정하여야 한다. 그러나 도형방식으로 GM_ADVANCED 를 설정한 경우에는 이 두 파라메터에 서로 다른 값을 설정할수 있다. 도형방식을 설정하는 방법은 제 9 장에서 설명한다.*

Weight 에는 0～100 의 범위에서 서체의 무게를 임의로 설정한다. 이 값을 령으로 하면 체계설정의 굵기가 설정된다. 표준의 굵기값은 400 이며 굵은체는 700 이다. 서체의 굵기를 설정하기 위해 아래에 보여 주는 마크로들중 어느 하나를 사용할수도 있다.

FW_DONTCARE
FW_THIN
FW_EXTRALIGHT
FW_LIGHT
FW_NORMAL
FW_MEDIUM
FW_SEMIBOLD
FW_BOLD
FW_EXTRABOLD
FW_HEAVY

경사체를 작성하는 경우에는 Ital 에 0 이 아닌 값을 설정한다. 경사체가 아닌 경우에는 이 파라메터에 령을 설정한다. 밑선이 있는 서체를 작성하는 경우에는 Underline 에 령 아닌 값을 설정한다. 밑선이 없는 경우에는 이 파라메터를 령으로 설정한다. 취소선이 있는 서체를 작성하는 경우에는 StrikeThru 에 0 아닌 값을 설정한다. 취소선이 없는 경우에는 이 파라메터에 령을 설정한다.

Charset 에는 *문자모임*을 설정한다. 뒤에서 작성하는 실례프로그람에서는 여기에 *ANSI_CHARSET* 를 설정하고 있다. Precision 에는 출력의 정밀도를 설정한다. 이 파라메터는 출력을 요구되는 서체의 실지 형태에 어느 정도 맞추겠는가를 결정한다. 실례프로그람에서는 *OUT_DEFAULT_PRECIS* 를 설정하고 있다.

ClipPrecision 에는 자르기(Clipping)의 정밀도를 설정한다. 이것은 문자가 자르기령역을 빠져나왔을 때 어떻게 자르기를 진행하는가를 결정하는것이다. 실례프로그람에서는 *CLIP_DEFAULT_PRECIS*를 설정하고 있다.(Charset, Precision 및 ClipPrecision 에 설정할수 있는 다른 값들에 대해서는 API 참고서를 참조하면 된다.)

Quality 는 실제의 출력장치를 위해 제공되는 물리서체에 론리서체를 어느 정도 맞추겠는가를 결정하는것이다. 일반적으로 사용되는 값을 아래에 보여 주었다.

DEFAULT_QUALITY　　DRAFT_QUALITY　　PROOF_QUALITY

PitchFam 에는 서체의 간격과 계렬을 설정한다. 간격으로는 다음의 어느 한 마크로를 선택할수 있다.

DEFAULT_PITCH　　FIXED_PITCH　　VARIABLE_PITCH

유효한 서체의 계렬은 다음의 여섯개이다.

FF_DECORATIVE　　FF_DONTCARE　　FF_MODERN
FF_ROMAN　　FF_SCRIPT　　FF_SWISS

서체계렬이 무엇 이든 상관 없는 경우에는 *FF_DONTCARE 계렬*을 설정한다. 서체의 계렬은 지정한 서체형이 체계에 존재하지 않는 경우에만 의미가 있다. PitchFam 의 값을 설정하려면 간격의 값과 계렬의 값을 하나씩 OR 연산으로 결합한다.

서체형이름의 지시자를 TypefaceName 에 설정한다. 이 이름은 32 문자이하여야 한다. 이 서체형이름의 서체가 체계에 설치되여 있어야 한다. 그러나 이 파라메터에 NULL 을 설정하면 다른 파라메터에서 설정한 외형과 호환성 있는 서체를 Windows 2000 이 자동적으로 선택하여 준다.(이 장의 뒤부분에서 사용가능한 서체를 렬거하는 방법을 설명한다.)

호출이 성공하면 *CreateFont( )*는 서체의 손잡이를 돌려 준다. 실패한 경우에는 NULL 을 돌려 준다. *CreateFont( )*를 사용하여 작성된 서체는 프로그람을 완료하기전에 삭제하여야 한다. 서체를 삭제하려면 *DeleteObject( )*를 호출한다.

두 종류의 전용서체 실례프로그람을 실례 8-3 에 보여 주었다. 첫번째 서체는 Courier New 에 기초한것이며 두번째 서체는 Century Gothic 에 기초한것이다. [Change Font]차림표를 선택하면 새로운 서체가 선택된다. 이 프로그람은 앞 절에서

작성한 프로그람과 같은 자원파일 및 같은 머리부파일을 리용한다. 프로그람의 실행결과를 그림 8-3에 보여 주었다.

실례 8-3. CustomFont 프로그람

```
// 전용서체의 작성

#include <windows.h>
#include <cstring>
#include <cstdio>
#include "text.h"

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);

char szWinName[] = "MyWin"; // 창문클라스의 이름

char str[255]; // 출력할 문자렬을 보관한다.
char fname[40] = "Default"; // 서체의 이름

int X=0, Y=0;   // 현재의 출력위치
int maxX, maxY; // 화면의 크기

HDC memdc;     // 가상장치의 손잡이를 보관한다.
HBITMAP hbit;  // 가상비트매프를 보관한다.
HBRUSH hbrush; // 붓의 손잡이를 보관한다.
HFONT holdf, hnewf1, hnewf2; // 서체의 손잡이를 보관한다.

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;
  HACCEL hAccel;

  // 창문클라스를 정의한다.
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst;     // 실체의 손잡이
  wcl.lpszClassName = szWinName; // 창문클라스의 이름
  wcl.lpfnWndProc = WindowFunc;  // 창문함수
```

```
wcl.style = 0;                  // 체계설정의 형식

wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

wcl.lpszMenuName = "FontMenu"; // 기본차림표

wcl.cbClsExtra = 0;       // 보조기억기령역은 필요 없다.
wcl.cbWndExtra = 0;

// 창문의 배경색을 흰색으로 한다.
wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

// 창문클라스를 등록한다.
if(!RegisterClassEx(&wcl)) return 0;

/* 창문클라스가 등록되였으므로
   창문을 작성할수 있다. */
hwnd = CreateWindow(
  szWinName, // 창문클라스의 이름
  "Using Custom Fonts", // 제목
  WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
  CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 너비는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 높이는 Windows가 결정하게 한다.
  NULL,          // 어미창문은 없다.
  NULL,          // 차림표는 없다.
  hThisInst,     // 실체의 손잡이
  NULL           // 추가파라메터는 없다.
);

// 건반가속기를 적재한다.
hAccel = LoadAccelerators(hThisInst, "FontMenu");

// 창문을 표시한다.
ShowWindow(hwnd, nWinMode);
UpdateWindow(hwnd);

// 통보문순환고리를 작성한다.
while(GetMessage(&msg, NULL, 0, 0))
```

```
    {
      if(!TranslateAccelerator(hwnd, hAccel, &msg)) {
        TranslateMessage(&msg); // 건반통보를 변환한다.
        DispatchMessage(&msg); // Windows 2000 에 조종을 넘긴다.
      }
    }

  return msg.wParam;
}

/* 이 함수는 Windows 2000 으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받아 들인다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  HDC hdc;
  PAINTSTRUCT paintstruct;
  static TEXTMETRIC tm;
  SIZE size;
  static fontswitch = 0;
  int response;

  switch(message) {
    case WM_CREATE:
      // 화면의 크기를 얻는다.
      maxX = GetSystemMetrics(SM_CXSCREEN);
      maxY = GetSystemMetrics(SM_CYSCREEN);

      // 가상창문을 작성한다.
      hdc = GetDC(hwnd);
      memdc = CreateCompatibleDC(hdc);
      hbit = CreateCompatibleBitmap(hdc, maxX, maxY);
      SelectObject(memdc, hbit);
      hbrush = (HBRUSH) GetStockObject(WHITE_BRUSH);
      SelectObject(memdc, hbrush);
      PatBlt(memdc, 0, 0, maxX, maxY, PATCOPY);

      // 새로운 서체를 작성한다.
      hnewf1 = CreateFont(14, 0, 0, 0, FW_NORMAL,
```

```
                    0, 0, 0, ANSI_CHARSET,
                    OUT_DEFAULT_PRECIS,
                    CLIP_DEFAULT_PRECIS,
                    DEFAULT_QUALITY,
                    DEFAULT_PITCH | FF_DONTCARE,
                    "Courier New");
  hnewf2 = CreateFont(20, 0, 0, 0, FW_SEMIBOLD,
                    0, 0, 0, ANSI_CHARSET,
                    OUT_DEFAULT_PRECIS,
                    CLIP_DEFAULT_PRECIS,
                    DEFAULT_QUALITY,
                    DEFAULT_PITCH | FF_DONTCARE,
                    "Century Gothic");
  ReleaseDC(hwnd, hdc);
  break;
case WM_COMMAND:
  switch(LOWORD(wParam)) {
    case IDM_SHOW:
      // 본문색을 검은색으로, 배경방식을 TRANSPARENT 로 한다.
      SetTextColor(memdc, RGB(0, 0, 0));
      SetBkMode(memdc, TRANSPARENT);

      // 본문치수를 얻는다.
      GetTextMetrics(memdc, &tm);

      sprintf(str, "%s font is %ld pixels high.",
              fname, tm.tmHeight);
      TextOut(memdc, X, Y, str, strlen(str));
      Y = Y + tm.tmHeight + tm.tmExternalLeading; // 개행한다.

      strcpy(str, "This is on the next line. ");
      TextOut(memdc, X, Y, str, strlen(str));

      // 문자렬의 길이를 구한다.
      GetTextExtentPoint32(memdc, str, strlen(str), &size);
      sprintf(str, "Previous string is %ld units long",
              size.cx);
      X = size.cx; // 앞 문자렬의 마감위치로 이동한다.
      TextOut(memdc, X, Y, str, strlen(str));
      Y = Y + tm.tmHeight + tm.tmExternalLeading; // 개행한다.
```

```
      X = 0; // X 자리표를 재설정한다.

      sprintf(str, "Screen dimensions: %d %d", maxX, maxY);
      TextOut(memdc, X, Y, str, strlen(str));
      Y = Y + tm.tmHeight + tm.tmExternalLeading; // 개행한다.

      InvalidateRect(hwnd, NULL, 1);
      break;
    case IDM_RESET:
      X = Y = 0;
      // 배경을 색칠하여 소거한다.
      PatBlt(memdc, 0, 0, maxX, maxY, PATCOPY);
      InvalidateRect(hwnd, NULL, 1);
      break;
    case IDM_FONT:
      switch(fontswitch) {
        case 0: // 새로운 서체 1로 반전절환한다.
          holdf = (HFONT) SelectObject(memdc, hnewf1);
          fontswitch = 1;
         strcpy(fname, "Courier New");
          break;
        case 1: // 새로운 서체 2로 반전절환한다.
          SelectObject(memdc, hnewf2);
          fontswitch = 2;
          strcpy(fname, "Century Gothic");
          break;
        default: // 본래의 서체로 반전절환한다.
          SelectObject(memdc, holdf);
          fontswitch = 0;
          strcpy(fname, "Default");
      }
      break;
    case IDM_EXIT:
      response = MessageBox(hwnd, "Quit the Program?",
                            "Exit", MB_YESNO);
      if(response == IDYES) PostQuitMessage(0);
      break;
    case IDM_HELP:
```

```
          MessageBox(hwnd,
                  "F2: Display\nF3: Change Font\nF4: Reset",
                  "Custom Fonts", MB_OK);
          break;
        }
        break;
      case WM_PAINT: // 다시그리기요구를 처리한다.
        hdc = BeginPaint(hwnd, &paintstruct); // 장치상황을 얻는다.

        // 가상창문을 화면에 복사한다.
        BitBlt(hdc, 0, 0, maxX, maxY, memdc, 0, 0, SRCCOPY);

        EndPaint(hwnd, &paintstruct); // 장치상황을 해제한다.
        break;
      case WM_DESTROY: // 프로그람을 끝낸다.
        DeleteDC(memdc);
        DeleteObject(hnewf1);
        DeleteObject(hnewf2);
        PostQuitMessage(0);
        break;
      default:
        /* 이 switch 문에서 지정된것 이외의 통보문은
           Windows 2000 에 처리를 맡긴다. */
        return DefWindowProc(hwnd, message, wParam, lParam);
    }

    return 0;
}
```

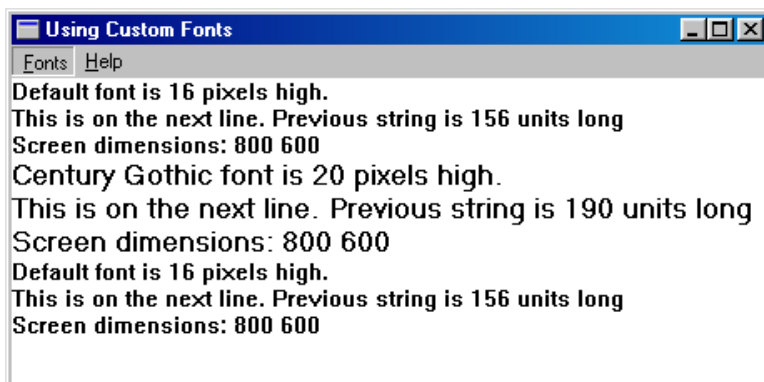



그림 8-3. 전용서체프로그람의 실행결과

CreateFontIndirect( )의 사용방법

CreateFont( )를 대신하여 여러가지 경우에 편리하게 쓰이는 *CreateFontIndirect( )* 라는 API 함수가 있다. 이 함수는 *LOGFONT* 구조체에 설정된 정보에 토대하여 전용서체를 작성한다. 아래에 선언을 보여 주었다.

HFONT CreateFontIndirect(CONST LOGFONT *lpFont);

이 함수는 lpFont 에서 가리키는 LOGFONT 구조체로 지정된 정보에 가장 가까운 서체를 작성하고 그의 손잡이를 돌려 준다. 함수의 호출이 실패한 경우는 NULL 을 돌려 준다. 프로그람을 실행하기전에 DeleteObject( )를 사용하여 서체의 손잡이를 삭제하여야 한다.

LOGFONT 구조체는 서체와 관련한 론리정보를 보관한다. 다음에 구조체의 정의를 보여 주었다.

```
typedef struct tagLOGFONT
{
            LONG lfHeight;                  // 서체의 높이
            LONG lfWidth;                   // 서체의 너비
            LONG lfEscapement;              // 본문의 각도
            LONG lfOrientation;             // 문자의 경사
            LONG lfWeight;                  // 굵기
            BYTE lfItalic;                  // 경사체라면 령 아닌 값
            BYTE lfUnderline;               // 밑선이 있으면 령 아닌 값
            BYTE lfStrikeOut;               // 취소선이 있으면 령 아닌 값
            BYTE lfCharSet;                 // 문자모임
            BYTE lfOutPrecision;            // 출력의 정밀도
            BYTE lfClipPrecision;           // 자르기의 정밀도
            BYTE lfQuality;                 // 출력의 질
            BYTE lfPitchAndFamily;          // 간격과 서체계렬
            CHAR lfFaceName[LF_FACESIZE];   // 이름
}   LOGFONT;
```

LOGFONT 구조체의 성원은 앞에서 설명한 CreateFont( )의 파라메터와 의미와 사용방법이 동일하다. 실례로 앞의 프로그람에서 CreateFont( )를 사용하여 작성한 Century Gothic 서체와 같은것을 CreateFontIndirect( )를 사용하여 작성하는 프로그람

코드를 아래에 보여 주었다.

```
LOGFONT lf;

lf.lfHeight = 20;
lf.lfWidth = 0;
lf.lfEscapement = 0;
lf.lfOrientation = 0;
lf.lfWeight = FW_SEMIBOLD;
lf.lfItalic = 0;
lf.lfUnderline = 0;
lf.lfStrikeOut = 0;
lf.lfCharSet = ANSI_CHARSET;
lf.lfOutPrecision = OUT_DEFAULT_PRECIS;
lf.lfClipPrecision = CLIP_DEFAULT_PRECIS;
lf.lfQuality = DEFAULT_QUALITY;
lf.lfPitchAndFamily = DEFAULT_PITCH | FF_DONTCARE;
strcopy(lf.lfFaceName, “Century Gothic”);
hnewf2 = CreateFontIndirect(&lf);
```

이 경우에는 CreateFont( )를 대신하여 CreateFontIndirect( )를 사용해야 할 필요는 없지만 반드시 CreatFontIndirect( )를 사용해야 하는 경우도 존재한다. 실례로 기존의 서체정보를 얻을 때는 서체의 속성이 LOGFONT 구조체에 보관된다. 이 구조체의 정보를 사용하여 CreateFontIndirect( )로 서체를 작성할수 있다.

본문의 회전

Windows 2000 이 제공하는 서체조종기능중 가장 우수한 기능의 하나로서 본문을 X 축의 주위로 회전시키거나 문자의 경사도를 간단히 변경시키는 기능이 있다. 이것은 화면상에 수평이 아닌 방향으로도 본문을 표시할수 있다는것이다.

본문출력의 각도를 바꾸려면 CreateFont( )를 사용하여 서체를 작성할 때 Escapement 와 Orientation 에 목적하는 각도를 설정한다. 이 각도는 수평축으로부터 1/10 도단위로 시계바늘회전방향의 반대방향으로 하여 설정한다. Escapement 는 본문의 기초선의 각도를 결정하고 Orientation 은 문자의 경사도를 결정한다.

본문을 회전시키려고 한다면 이 두개의 값들을 같게 하여야 한다. 이렇게 하면 회전

된 기초선을 따라 문자렬이 놓이게 된다.

실례 8-4의 프로그람은 앞의 프로그람에서 작성된 Courier와 Gothic 서체를 변경하여 그것들이 45° 경사로 표시되도록 한다. Courier 서체의 본문은 Escapement 와 Orientation 에 정의값을 주므로 오른쪽 웃방향으로 표시된다. Gothic 서체의 본문은 Escapement 와 Orientation 에 부의값을 주고 있으므로 오른쪽 아래방향으로 표시된다. 프로그람의 실행결과를 그림 8-4에 보여 주었다.

실례 8-4. RotateFont 프로그람

```
// 본문의 회전

#include <windows.h>
#include <cstring>
#include <cstdio>

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);

char szWinName[] = "MyWin"; // 창문클라스의 이름

char str[255]; //  출력할 문자렬을 보관한다.

HFONT hnewf1, hnewf2; // 서체의 손잡이를 보관한다.

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;

  // 창문클라스를 정의한다.
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst;      // 실체의 손잡이
  wcl.lpszClassName = szWinName; // 창문클라스의 이름
  wcl.lpfnWndProc = WindowFunc;  // 창문함수
  wcl.style = 0;                 // 체계설정의 형식
```

```
wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

wcl.lpszMenuName = NULL; // 클라스차림표는 없다.
wcl.cbClsExtra = 0;      // 보조기억기령역은 필요 없다.
wcl.cbWndExtra = 0;

// 창문의 배경색을 흰색으로 한다.
wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

// 창문클라스를 등록한다.
if(!RegisterClassEx(&wcl)) return 0;

/* 창문클라스가 등록되였으므로
   창문을 작성할수 있다. */
hwnd = CreateWindow(
  szWinName, // 창문클라스의 이름
  "Rotating Text", // 제목
  WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
  CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 너비는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 높이는 Windows가 결정하게 한다.
  NULL,          // 어미창문은 없다.
  NULL,          // 차림표는 없다.
  hThisInst,     // 실체의 손잡이
  NULL           // 추가파라메터는 없다.
);

// 창문을 표시한다.
ShowWindow(hwnd, nWinMode);
UpdateWindow(hwnd);

// 통보문순환고리를 작성한다.
while(GetMessage(&msg, NULL, 0, 0))
{
  TranslateMessage(&msg); // 건반통보를 변환한다.
```

```
        DispatchMessage(&msg);  // Windows 2000 에 조종을 넘긴다.
    }
    return msg.wParam;
}

/* 이 함수는 Windows 2000 으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받아 들인다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
    HDC hdc;
    PAINTSTRUCT ps;

    switch(message) {
        case WM_CREATE:
            // 이 서체의 각도는 45° 오른쪽웃방향이다.
            hnewf1 = CreateFont(14, 0, 450, 450, FW_NORMAL,
                                0, 0, 0, ANSI_CHARSET,
                                OUT_DEFAULT_PRECIS,
                                CLIP_DEFAULT_PRECIS,
                                DEFAULT_QUALITY,
                                DEFAULT_PITCH | FF_DONTCARE,
                                "Courier New");

            // 이 서체의 각도는 45° 오른쪽웃방향이다.
            hnewf2 = CreateFont(20, 0, -450, -450, FW_SEMIBOLD,
                                0, 0, 0, ANSI_CHARSET,
                                OUT_DEFAULT_PRECIS,
                                CLIP_DEFAULT_PRECIS,
                                DEFAULT_QUALITY,
                                DEFAULT_PITCH | FF_DONTCARE,
                                "Century Gothic");
            break;
        case WM_PAINT: // 다시그리기요구를 처리한다.
            hdc = BeginPaint(hwnd, &ps); // 장치상황을 얻는다.

            SelectObject(hdc, hnewf1);
```

```
        strcpy(str, "This string is angled 45 degrees up.");
        TextOut(hdc, 0, 200, str, strlen(str));

        SelectObject(hdc, hnewf2);
        strcpy(str, "This string is angled 45 degrees down.");
        TextOut(hdc, 10, 0, str, strlen(str));

        EndPaint(hwnd, &ps); // 장치상황을 해제한다.
        break;
      case WM_DESTROY: // 장치상황을 해제하고 프로그람을 끝낸다.
        DeleteObject(hnewf1);
        DeleteObject(hnewf2);
        PostQuitMessage(0);
        break;
      default:
        /* 이 switch 문에서 지정된것 이외의 통보문은
           Windows 2000 에 처리를 맡긴다. */
        return DefWindowProc(hwnd, message, wParam, lParam);
    }
    return 0;
}
```

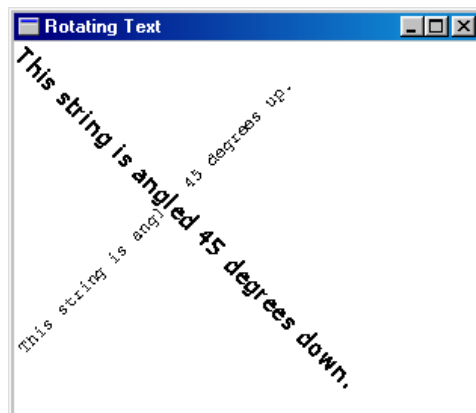



그림 8-4. 본문의 회전

서체의 렬거

지금까지의 프로그람에서 서체를 작성할 때는 목적하는 서체를 사용할수 있다는것을 가정하고 있었다. 그러나 Windows 프로그람을 작성하면서 어떤 상황을 가정하는것은 좋지 않은 일이다. 례하면 체계에 서체가 추가될수도 있고 체계로부터 서체가 삭제될수도 있다.

사용가능한 서체를 렬거하려면 *EnumFontFamiliesEx( )*라는 API 함수를 사용한다. 아래에 선언을 보여 주었다.

```
int EnumFontFamiliesEx(HDC hdc, LPLOGFONT lpFontInfo,
                       FONTENUMPROC EnumFunc,
                       LPARAM lParam, DWORD NotUsed);
```

hdc는 서체를 얻는 장치상황의 손잡이이다. 장치상황이 다르면 지원되는 서체도 다르다. 서체의 종류를 정의하는 다양한 속성은 lpFontInfo( )에서 지정되는 LOGFONT 구조체에 보관된다.(LOGFONT 구조체에 대해서는 이 장을 시작하면서 설명하였다.)

EnumFunc는 서체가 하나 렬거될 때마다 호출되는 역호출함수에 대한 지시자이다. lParam 은 EnumFunc 에서 지정되는 역호출함수에 응용프로그람으로부터 어떤 정보를 보내는데 사용된다. NotUsed 는 사용되지 않으므로 령을 설정하여야 한다. 이 함수는 EnumFunc가 맨 마지막에 호출되였을 때의 돌림값을 돌려 준다.

EnumFontFamiliesEx( )을 호출하기전에 lpFontInfo에서 지정되는 LOGFONT 구조체의 세개 성원 lfCharSet, lfPitchAndFamily 및 lfFaceName을 초기화하여야 한다.

특정한 문자모임의 서체만을 렬거하는 경우에는 lfCharSet 에 문자모임의 이름을 설정한다. 모든 문자모임의 서체를 렬거하는 경우에는 *DEFAULT_CHARSET* 를 설정한다. 특정한 서체형의 서체만을 렬거하는 경우에는 lfFaceName에 서체형의 이름을 설정한다. 모든 서체형의 서체를 렬거하는 경우에는 이 성원에 령문자렬을 설정한다. lfPitchAndFamily에는 령을 설정하여야 한다.

Windows 2000 이 서체를 렬거할 때마다 EnumFunc 에서 지정된 함수가 호출된다. 이 함수는 렬거된 서체를 처리하며 계속하여 다른 서체를 요구한다면 령 아닌 값을 돌려주며 요구하지 않는다면 령을 돌려 준다. 이 함수의 정의는 다음과 같다.

```
int CALLBACK EnumFunc(ENUMLOGFONTEX *lpLFInfo,
                 NEWTEXTMETRICEX *lpTMinfo,
                 int type, LPARAM lParam);
```

lpLFInfo 는 렬거된 서체에 대한 론리정보를 보관하는 ENUMLOGFONTEX 구조체

의 지시자이다. lpTMinfo 는 서체의 물리정보를 보관하는 NEWTEXTMETRICEX 구조체의 지시자이다.

TrueType 가 아닌 서체의 경우에는 NEWTEXTMETRICEX 구조체 대신에 TEXTMETRIC 구조체의 지시자가 전해 진다. type 는 서체의 종류를 가리키는것이므로 다음의 어느 한 값으로 된다.

RASTER_FONTTYPE　　　TRUETYPE_FONTTYPE
DEVICE_FONTTYPE

lParam 에는 Ex( )의 lParam 에 설정된 값이 보관된다.

이식과 관련한 요점 : Windows 3.1 은 EnumFontFamiliesEx( )를 지원하지 않는다. Windows 3.1 프로그람에서는 EnumFont( ) 또는 EnumFontFamilies( )를 사용한다. Windows 2000 에로 이식하는 경우에는 이것들을 EnumFontFamiliesEx( )로 치환해야 한다.

ENUMLOGFONTEX 구조체의 정의를 아래에 보여 주었다.

```
typedef struct tagENUMLOGFONTEX
{
    LOGFONT elfLogFont;
    BYTE elfFullName[LF_FULLFACESIZE];    // 서체의 완전이름
    BYTE elfStyle[LF_FACESIZE];           // 서체의 형식
    BYTE elfScript[LF_FACESIZE];          // 서체에서 사용되는 대본
} ENUMLOGFONTEX;
```

elfLogFont 는 론리서체의 거의 모든 정보를 가지고 있는 LOGFONT 구조체이다. 이 구조체가 가지는 정보는 CreateFont( ) 혹은 CreateFontIndirect( )를 호출할 때 사용된다. 서체의 완전한 이름은 elfFullName 에 보관된다. 형식(굵은체, 경사체 등)은 elfStyle 에 보관된다. 이 성원은 TrueType 서체가 아닌 경우에는 의미가 없다. 대본(Script)의 이름은 elfScript 에 보관된다.

서체렬거프로그람

실례 8-5 의 프로그람코드는 두가지 방법으로 서체를 렬거하는 방법을 보여 주었다. 먼저 [Enumerate]차림표에서 [Available Font]항목을 선택하면 지원되는 모든 서체형의 서체가 렬거된다. [Selected Typeface]를 선택하면 지정된 서체형의 서체만이 렬거된다. 프로그람의 실행결과를 그림 8-5 에 보여 주었다.

실례 8-5. Font 프로그람

```
// 서체의 렬거

#include <windows.h>
#include <cstring>
#include <cstdio>
#include "font.h"

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
int CALLBACK FontFunc(ENUMLOGFONTEX *lpLF,
                      NEWTEXTMETRICEX *lpTM,
                      int type, LPARAM lParam);
BOOL CALLBACK FontDialog(HWND hdwnd, UINT message,
                         WPARAM wParam, LPARAM lParam);

char szWinName[] = "MyWin"; // 창문클라스의 이름

char str[255];     // 출력할 문자렬을 보관한다.
char fontstr[255]; // 사용자가 입력한 서체이름을 보관한다.

int X=0, Y=0;    // 현재의 출력위치
int maxX, maxY; // 화면의 크기

int linespacing; // 행간격

HDC memdc;      // 가상장치의 손잡이를 보관한다.
HBITMAP hbit;  // 가상비트매프의 손잡이를 보관한다.
HBRUSH hbrush; // 붓의 손잡이를 보관한다.

HINSTANCE hInst;

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HWND hwnd;
  MSG msg;
```

```
WNDCLASSEX wcl;
HACCEL hAccel;

// 창문클라스를 정의한다.
wcl.cbSize = sizeof(WNDCLASSEX);

wcl.hInstance = hThisInst;      // 실체의 손잡이
wcl.lpszClassName = szWinName; // 창문클라스의 이름
wcl.lpfnWndProc = WindowFunc;  // 창문함수
wcl.style = 0;                  // 체계설정의 형식

wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

wcl.lpszMenuName = "FontEnumMenu"; // 기본차림표

wcl.cbClsExtra = 0; // 보조기억기령역은 필요 없다.
wcl.cbWndExtra = 0;

// 창문의 배경색을 흰색으로 한다.
wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

// 창문클라스를 등록한다.
if(!RegisterClassEx(&wcl)) return 0;

/* 창문클라스가 등록되였으므로
   창문을 작성할수 있다. */
hwnd = CreateWindow(
  szWinName, // 창문클라스의 이름
  "Enumerating Fonts", // 제목
  WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
  CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 너비는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 높이는 Windows가 결정하게 한다.
  NULL,          // 어미창문은 없다.
  NULL,          // 차림표는 없다.
```

```
    hThisInst,      // 실체의 손잡이
    NULL             // 추가파라메터는 없다.
  );

  hInst = hThisInst; // 현재 실체의 손잡이를 보관한다.

  // 건반가속기를 적재한다.
  hAccel = LoadAccelerators(hThisInst, "FontEnumMenu");

  // 창문을 표시한다.
  ShowWindow(hwnd, nWinMode);
  UpdateWindow(hwnd);

  // 통보문순환고리를 작성한다.
  while(GetMessage(&msg, NULL, 0, 0))
  {
    if(!TranslateAccelerator(hwnd, hAccel, &msg)) {
      TranslateMessage(&msg); // 건반통보를 변환한다.
      DispatchMessage(&msg); // Windows 2000에 조종을 넘긴다.
    }
  }

  return msg.wParam;
}

/* 이 함수는 Windows 2000으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받아 들인다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  HDC hdc;
  static TEXTMETRIC tm;
  PAINTSTRUCT ps;
  LOGFONT lf;
  int result;
  int response;
```

```
    switch(message) {
      case WM_CREATE:
        // 화면의 크기를 얻는다.
        maxX = GetSystemMetrics(SM_CXSCREEN);
        maxY = GetSystemMetrics(SM_CYSCREEN);

        // 가상창문을 작성한다.
        hdc = GetDC(hwnd);
        memdc = CreateCompatibleDC(hdc);
        hbit = CreateCompatibleBitmap(hdc, maxX, maxY);
        SelectObject(memdc, hbit);
        hbrush = (HBRUSH) GetStockObject(WHITE_BRUSH);
        SelectObject(memdc, hbrush);
        PatBlt(memdc, 0, 0, maxX, maxY, PATCOPY);

        // 본문치수를 얻는다.
        GetTextMetrics(memdc, &tm);
        // 행간격을 구한다.
        linespacing = tm.tmHeight + tm.tmExternalLeading;

        // 귤색으로 표식을 표시한다.
        SetTextColor(memdc, RGB(255, 100, 0));
        TextOut(memdc, X, 0, "Typeface", strlen("Typeface"));
        TextOut(memdc, X+200, 0, "Style", strlen("Style"));
        TextOut(memdc, X+300, 0, "Script", strlen("Script"));
        SetTextColor(memdc, RGB(0, 0, 0));

        ReleaseDC(hwnd, hdc);
        break;
     case WM_COMMAND:
        switch(LOWORD(wParam)) {
          case IDM_FONTS: // 서체를 표시한다.
            Y = linespacing + linespacing/2;
            PatBlt(memdc, 0, linespacing, maxX, maxY, PATCOPY);

            lf.lfCharSet = DEFAULT_CHARSET;
            strcpy(lf.lfFaceName, "");
            lf.lfPitchAndFamily = 0;
```

```
    // 서체를 렬거한다.
    hdc = GetDC(hwnd);
    EnumFontFamiliesEx(hdc, &lf,
        (FONTENUMPROC) FontFunc, (LPARAM)hwnd, 0);
    ReleaseDC(hwnd, hdc);

    break;
  case IDM_TYPEFACE: // 선택된 서체형을 표시한다.
    // 서체의 이름을 얻는다.
    result = DialogBox(hInst, "FontDB", hwnd,
                       (DLGPROC) FontDialog);

    if(!result) break; // 사용자가 취소하였다.

    Y = linespacing + linespacing/2;
    PatBlt(memdc, 0, linespacing, maxX, maxY, PATCOPY);

    lf.lfCharSet = DEFAULT_CHARSET;
    strcpy(lf.lfFaceName, fontstr);
    lf.lfPitchAndFamily = 0;

    // 지정된 서체의 모든 형식을 렬거한다.
    hdc = GetDC(hwnd);
    EnumFontFamiliesEx(hdc, &lf,
        (FONTENUMPROC) FontFunc, (LPARAM) hwnd, 0);
    ReleaseDC(hwnd, hdc);

    break;
  case IDM_EXIT:
    response = MessageBox(hwnd, "Quit the Program?",
                          "Exit", MB_YESNO);
    if(response == IDYES) PostQuitMessage(0);
    break;
  case IDM_HELP:
    MessageBox(hwnd,
               "F2: Show Fonts\nF3: Show Typeface\n"
               "F3: Exit", "Show Fonts", MB_OK);
```

```
            break;
        }
        break;
      case WM_PAINT: // 다시그리기요구를 처리한다.
        hdc = BeginPaint(hwnd, &ps); // 장치상황을 얻는다.

        // 가상창문을 화면에 복사한다.
        BitBlt(hdc, 0, 0, maxX, maxY, memdc, 0, 0, SRCCOPY);
        EndPaint(hwnd, &ps); // 장치상황을 해제한다.
        break;
      case WM_DESTROY: // 프로그람을 끝낸다.
        DeleteDC(memdc);
        PostQuitMessage(0);
        break;
      default:
        /* 이 switch 문에서 지정된것 이외의 통보문은
           Windows 2000 에 처리를 맡긴다. */
        return DefWindowProc(hwnd, message, wParam, lParam);
    }

    return 0;
}

// 서체를 렬거하는 역호출함수
int CALLBACK FontFunc(ENUMLOGFONTEX *lpLF,
                      NEWTEXTMETRICEX *lpTM,
                      int type, LPARAM lParam)
{
    int response;
    RECT rect;

    // 서체의 정보를 표시한다.
    TextOut(memdc, X, Y, lpLF->elfLogFont.lfFaceName,
            strlen(lpLF->elfLogFont.lfFaceName)); // 서체의 이름

    if(type == TRUETYPE_FONTTYPE)
      TextOut(memdc, X+200, Y, (char *)lpLF->elfStyle,
              strlen((char *)lpLF->elfStyle)); // 형식
```

```
    else
      TextOut(memdc, X+200, Y, "N/A", 3);

    TextOut(memdc, X+300, Y, (char *)lpLF->elfScript,
            strlen((char *)lpLF->elfScript)); // 대본의 종류

    Y += linespacing;

    InvalidateRect((HWND)lParam, NULL, 0);

    // 의뢰자구역의 현재 크기를 얻는다.
    GetClientRect((HWND)lParam, &rect);

    // 창문의 하단에서 정지한다.
    if( (Y + linespacing) >= rect.bottom) {
      Y = linespacing + linespacing/2; // 창문의 상단으로 돌아간다.
      response = MessageBox((HWND)lParam, "More?",
                            "More Fonts?", MB_YESNO);
      if(response == IDNO) return 0;
      PatBlt(memdc, 0, linespacing, maxX, maxY, PATCOPY);
    }

    return 1;
  }

  // 서체를 렬거하는 대화칸
  BOOL CALLBACK FontDialog(HWND hdwnd, UINT message,
                           WPARAM wParam, LPARAM lParam)
  {
    switch(message) {
      case WM_COMMAND:
        switch(LOWORD(wParam)) {
          case IDCANCEL:
            EndDialog(hdwnd, 0);
            return 1;
          case IDD_ENUM:
            // 서체형의 이름을 얻는다.
            GetDlgItemText(hdwnd, IDD_EB1, fontstr, 80);
```

```
            EndDialog(hdwnd, 1);
            return 1;
        }
        break;
    }

    return 0;
}
```

이 프로그람은 아래의 자원파일을 리용한다.

```
#include <windows.h>
#include "font.h"

FontEnumMenu MENU
{
  POPUP "Enumerate" {
    MENUITEM "Available &Fonts\tF2", IDM_FONTS
    MENUITEM "Selected &Typeface\tF3", IDM_TYPEFACE
    MENUITEM "E&xit\tCtrl+X", IDM_EXIT
  }
  MENUITEM "&Help", IDM_HELP
}

FontEnumMenu ACCELERATORS
{
  VK_F1, IDM_HELP, VIRTKEY
  VK_F2, IDM_FONTS, VIRTKEY
  VK_F3, IDM_TYPEFACE, VIRTKEY
  "^X", IDM_EXIT
}

FontDB DIALOGEX 10, 10, 100, 60
CAPTION "Enumerate Typeface"
STYLE WS_POPUP | WS_CAPTION | WS_SYSMENU
{
  CTEXT "Enter Typeface", 300, 10, 10, 80, 12
  EDITTEXT IDD_EB1, 10, 20, 80, 12, ES_LEFT |
```

```
            WS_BORDER | ES_AUTOHSCROLL | WS_TABSTOP
    DEFPUSHBUTTON "Enumerate" IDD_ENUM, 30, 40, 40, 14
  }
```

머리부파일 FONT.H의 내용을 아래에 보여 주었다.

```
  #define IDM_FONTS          100
  #define IDM_TYPEFACE       101
  #define IDM_EXIT           102
  #define IDM_HELP           103

  #define IDD_EB1            200
  #define IDD_ENUM           201
```

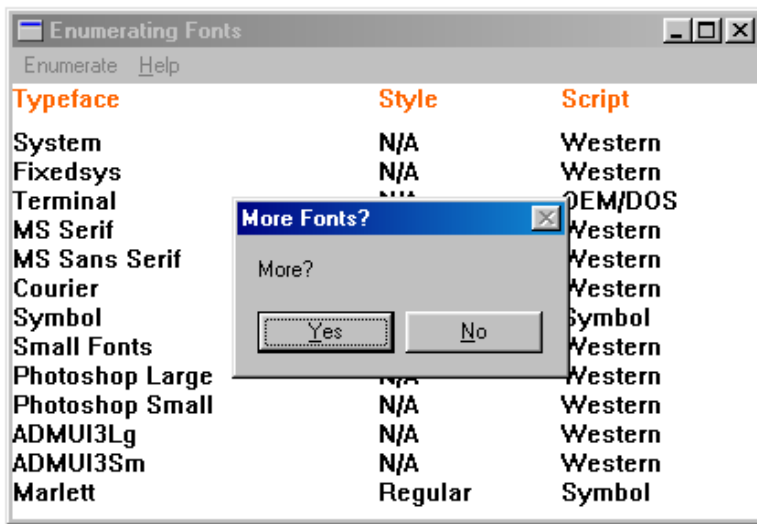



그림 8-5. 서체렬거프로그람의 실행결과

프로그람의 내용을 자세히 살펴 보자. [Available]차림표가 선택되면 lf.lfFaceName에 NULL을 설정하여 EnumFontFamiliesEx()가 호출된다. 이에 의해 체계가 지원하는 모든 서체형의 서체가 렬거된다.

[Selected Typeface]차림표가 선택되면 작은 대화칸이 표시되고 거기에서 특정한 서체형의 이름을 입력할수 있다. 입력된 이름은 lf.lfFaceName 성원에 보관된다. 이렇게 되여 지정된 서체형의 서체만이 렬거된다. 프로그람의 처리를 간단히 하기 위해 이 함수는 창문을 다 채우게 정보를 표시하면 렬거를 계속하겠는가를 질문하는 통보칸을 표

시한다. 렬거를 계속하려는 경우에는 창문에 계속표시가 진행된다. 창문의 현재의 크기를 판정하기 위해 FontFunc( )는 *GetClientRect( )*를 호출하고 있다. *GetClientRect( )* 함수의 선언은 다음과 같다.

```
BOOL GetClientRect(HWND hwnd, LPRECT lpDim);
```

hwnd는 크기를 구하려는 창문의 손잡이이며 lpDim은 창문의 의뢰자구역의 현재의 크기가 보관되는 RECT 구조체의 지시자이다.(RECT 구조체는 제 3장에서 설명하였다.) 함수의 호출이 성공하면 령 아닌 값이 돌려 지며 실패하면 령이 돌려 진다. 이 함수는 창문의 의뢰자구역의 크기를 알려고 할 때 가장 편리하게 사용된다.

이 프로그람에서는 창문의 현재의 높이에 본문의 다음 행을 표시하는데 충분한 령역이 있는가 어떤가를 판정하기 위해 rect.bottom의 값을 사용하고 있다.

서체가 렬거될 때마다 LOGFONT 구조체에 서체의 정보가 보관된다. 이 구조체를 CreateFontIndirect( )에 리용하여 렬거된 매 서체를 작성하고 그 서체로 문자렬을 표시해 볼수 있다. 이렇게 하면 목적하는 서체의 형태를 눈으로 보면서 실제로 확인할수 있다.

Windows 2000이 지원하는 서체와 본문의 조종기능은 매우 풍부하다. 필요한 기능들에 대하여서는 프로그람작성자들 자신이 보다 자세히 조사해 보아야 할것이다. 다음 장에서는 도형을 주제로 하여 창문의 출력에 대한 설명을 계속한다.

제 9 장

도형의 사용법

Windows 2000 은 창문에 도형을 그리기 위한 API 함수들을 수많이 제공하고 있다. Windows 는 도형방식의 조작체계이므로 이것은 놀라운 일이 아니다. 도형은 Windows 환경의 모든 부분에 통합되여 있으므로 간단하게 취급할수 있다.

Windows 2000 이 지원하는 모든 도형함수에 대해 설명하는것은 불가능하므로 이 장에서는 그중에서 가장 기초적인 점, 직선, 4 각형 및 타원 등을 그리는 함수만을 설명한다. 이 장에서는 *도형*을 창문에 출력하는 방식을 변경하는 방법에 대해서도 설명한다. Windows 2000 의 도형기능은 매우 강력하므로 보다 심도 있게 알려면 직접 시험해 보아야 한다.

도형함수의 리용방법을 보여 주는 실례로서 간단한 그림그리기프로그람을 작성한다. Windows 2000 의 강력한 도형그리기기능에 의거하면 그리 길지 않은 프로그람코드로도 그림그리기프로그람을 작성할수 있다는것을 알게 될것이다.

도형의 자리표계

도형의 자리표계는 본문에 기초한 함수에서 사용하던것과 같다. 체계설정으로는 창문의 왼쪽웃모서리의 자리표가 원점인 (0, 0)으로 되며 론리단위는 화소(pixel)이다. 그러나 자리표계 및 론리단위로부터 화소에로 넘기는 방법은 임의로 변경할수 있다.

Windows 2000 및 Windows 전반은 *현재위치*(Current position)를 관리한다. 현재위치는 일부 도형함수들에서 참조되거나 변경된다. 프로그람의 기동시에는 현재위치가 (0, 0)으로 설정된다. 현재위치를 눈으로 볼수는 없으며 도형의 유표와 같은것은 표시되지 않는다. 현재위치는 도형함수가 다음에 그리기를 개시하는 창문상에서의 위치를 가리킨다.

펜과 붓

Windows 의 도형체계는 *펜*과 *붓*이라는 두개의 객체에 기초하고 있다. 4 각형이나 타원과 같은 닫긴 도형은 체계설정으로 현재 선택되여 있는 붓으로 채색된다. 펜은 직선이나 곡선을 그리기 위한 객체로서 여러가지 도형함수에서 사용된다. 체계설정의 펜은 검은색이며 한 화소의 너비로 되여 있으나 이러한 속성들을 변경시킬수 있다.

지금까지의 실례프로그람들에서는 내장된 객체만을 사용하여 왔다. 실례로 창문의 의뢰자구역을 색칠하는데 사용된 흰색의 내장붓을 들수 있다. 이 장에서는 전용붓이나 펜을 작성하는 방법에 대해서도 설명한다.

점의 그리기

*SetPixel()*이라는 API 함수를 사용하면 임의의 색으로 점을 그릴수 있다. 선언은 다음과 같다.

```
COLORREF SetPixel(HDC hdc, int X, int Y, COLORREF color);
```

hdc 는 그리기대상으로 되는 장치상황의 손잡이이다. 점의 자리표를 X, Y 에 설정하고 점의 색을 color 에 설정한다.(COLORREF 자료형은 제 8 장에서 설명하였다.) 이 함수는 호출직전의 점의 색을 돌려 주며 오유가 발생하거나 지정한 위치가 창문밖에 놓이는 경우에는 -1 을 돌려 준다.

직선의 그리기

직선을 그리자면 *LineTo( )*함수를 사용해야 한다. 이 함수는 현재 선택된 펜으로 직선을 그린다. 선언은 다음과 같다.

```
BOOL LineTo(HDC hdc, int X, int Y);
```

hdc에는 장치상황의 손잡이를 설정한다. 직선은 현재위치를 시작점으로 하고 X, Y로 지정된 자리표를 끝점으로 하여 그려진다. 그리기가 끝난후 현재위치는 (X, Y)로 변경된다. 이 함수는 호출이 성공하면 즉 직선이 그려 지면 령이 아닌 값을 돌려 주며 실패하면 령을 돌려 준다.

LineTo( )가 직선의 시작점으로 현재위치를 사용하고 그리기가 끝나면 현재위치를 직선의 끝점으로 변화시키는것은 흔히 직선을 그릴 때 방금 전에 그린 직선의 끝점에서 다음 직선의 그리기를 시작하는 경우가 많이 있기때문이다. 이러한 때 LineTo( )를 효과적으로 사용하여 시작점의 자리표를 다시 설정하는 번거로움을 배제할수 있다. 이것을 바라지 않는 다면 LineTo( )를 호출하기전에 다음에 설명하는 *MoveToEx( )*함수를 사용하여 임의의 위치에 현재위치를 설정할수 있다.

현재위치의 설정

임의의 위치에 *현재위치*를 설정하려면 *MoveToEx( )*함수를 사용해야 한다. 선언은 다음과 같다.

```
BOOL MoveToEx(HDC hdc, int X, int Y, LPPOINT lpCoord);
```

장치상황의 손잡이를 hdc에 설정하고 새로운 현재위치의 자리표를 X, Y에 설정한다. 직선의 현재위치가 lpCoord로 지적된 *POINT*구조체에 돌려 진다. POINT구조체의 정의는 다음과 같다.

```
typedef struct tagPOINT {
    LONG x;
    LONG y;
} POINT;
```

그러나 lpCoord 파라메터에 NULL을 설정한 경우에 MoveToEx( )는 현재위치를

돌려 주지 않는다.

MoveToEx( )함수는 호출이 성공하면 령이 아닌 값을 돌려 주고 실패하면 령을 돌려 준다.

원호의 그리기

*Arc( )함수*를 사용하면 현재펜의 색으로 원호(타원의 일부)를 그릴수 있다. 선언은 다음과 같다.

```
BOOL Arc(HDC hdc, int upX, int upY, int lowX, int lowY,
         int startX, int startY, int endX, int endY);
```

hdc 는 원호를 그릴 대상으로 되는 장치상황의 손잡이이다. 원호는 두개의 객체에 의해 정의된다. 원호는 4 각형으로 둘러싸인 타원의 일부이다.

이 4 각형의 왼쪽웃모서리는 upX, upY 로 지정되고 오른쪽아래모서리는 lowX, lowY 로 지정된다. 실제로 그려 지는 타원의 일부분 즉 원호는 4 각형의 중심과 startX, startY 로 지정되는 점을 련결하는 직선과의 사귐점에서부터 시작하여 4 각형의 중심과 endX, endY 로 지정된 점을 련결하는 직선과의 사귐점에서 끝난다.

체계설정으로는 원호가 startX, startY 로부터 시계바늘반대방향으로 그려 진다. 이 그리기방향을 *SetArcDirection( )*이라는 API 함수를 사용하여 변경시킬수도 있다. 그림 9-1 에 Arc( )의 동작을 보여 주었다.

Arc( )는 호출이 성공하면 령이 아닌 값을 돌려 주며 실패하면 령을 돌려 준다.

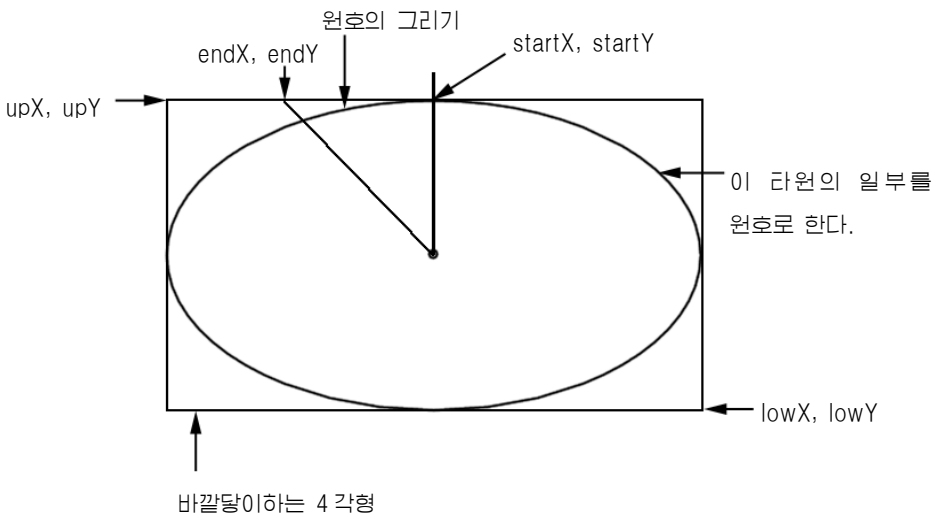



그림 9-1. Arc( )함수의 동작

4각형의 그리기

*Rectangle()*함수를 사용하면 현재펜의 색으로 4각형을 그릴수 있다. 선언은 다음과 같다.

BOOL Rectangle(HDC hdc, int upX, int upY, int lowX, int lowY);

hdc는 장치상황의 손잡이이다. 4각형의 왼쪽웃모서리의 자리표를 upX, upY에 설정하고 오른쪽웃모서리의 자리표를 lowX, lowY에 설정한다. 이 함수는 호출이 성공하면 령이 아닌 값을 돌려 주고 오유가 발생하면 령을 돌려 준다. 4 각형의 내부는 현재의 붓으로 색칠된다.

RoundRect( )함수를 사용하면 모서리가 원활한 4각형을 그릴수 있다. 모서리가 원활한 4각형이란 모서리부분만이 원형인 4각형이다. *RoundRect( )*함수의 선언은 다음과 같다.

BOOL RoundRect(HDC hdc, int upX, int upY, int lowX, int lowY,
int curveX, int curveY);

첫 다섯개의 파라메터는 Rectangle( )과 같다. 모서리를 원활하게 하는 곡선은 curveX, curveY 의 값으로 결정된다. 이 파라메터들은 곡선으로 되는 타원의 너비와 높이를 지정하기 위한것들이다. RoundRect( )는 호출이 성공하면 령이 아닌 값을 돌려주고 실패하면 령을 돌려 준다. 모서리가 원활한 4 각형의 내부는 현재 붓으로 자동적으로 채색된다.

타원과 부채형의 그리기

현재 선택되여 있는 펜의 색으로 타원이나 원을 그리자면 *Ellipse( )*함수를 사용하여야 한다. 선언은 다음과 같다.

BOOL Ellipse(HDC hdc, int upX, int upY, int lowX, int lowY);

hdc는 타원을 그리려는 장치상황이다. 타원은 바깥닿이하는 4각형을 지정하여 정의된다. 4각형의 왼쪽웃모서리를 upX, upY에 설정하고 오른쪽아래모서리를 lowX, lowY에 설정한다. 원을 그리는 경우에는 바른 4각형을 지정한다.

이 함수는 호출이 성공하면 령이 아닌 값을 돌려 주며 실패하면 령을 돌려 준다. 타원의 내부는 현재의 붓을 사용하여 색칠된다.

타원과 관련된 도형으로 부채형이 있다. 부채형은 원호 및 원호의 량단과 타원의 중심을 련결하는 직선으로 구성되는 객체이다. 부채형을 그리자면 *Pie( )*함수를 사용해야 한다. 선언은 다음과 같다.

```
BOOL Pie(HDC hdc, int upX, int upY, int lowX, int lowY,
         int startX, int startY,int endX, int endY);
```

hdc는 부채형이 그려 지는 장치상황의 손잡이이다. 원호부분은 두개의 객체에 의해 정의된다. 원호는 4각형으로 둘러싸인 타원의 일부분이다. 이 4각형의 왼쪽웃모서리는 upX, upY로 지정되고 오른쪽아래모서리의 자리표는 lowX, lowY로 지정된다.

실제로 그려 지는 타원의 일부 즉 원호는 4각형의 중심과 startX, startY로 지정되는 점을 련결하는 직선과의 사귐점으로부터 시작하여 4각형의 중심과 endX, endY로 지정된 점을 련결하는 직선과의 사귐점에서 끝난다. 체계설정으로는 원호가 시계바늘회전방향의 반대방향으로 그려진다.

부채형은 현재의 펜의 색으로 그려 지며 현재의 붓을 사용하여 채색된다. Pie( )함수는 호출이 성공하면 령이 아닌 값을 돌려 주며 오유가 발생하면 령을 돌려 준다.

펜의 조종

도형객체는 현재 선택된 *펜*을 사용하여 그릴수 있다. 체계설정의 펜은 검은색이며 너비는 1화소(Pixel)로 되여 있다. 내장펜에는 검은색, 흰색 및 빈 펜의 세 종류가 있다. 내장펜의 손잡이를 얻자면 이미 설명한 GetStockObject( )를 사용한다. 내장펜을 가리키는 마크로는 BLACK_PEN, WHITE_PEN 및 NULL_PEN이다. 펜손잡이의 자료형은 HPEN이다.

내장펜만으로는 실현할수 있는것이 제한되여 있으므로 응용프로그람에서 자체의 펜이 필요하게 된다. *CreatePen( )*을 사용하면 자체의 펜을 작성할수 있다. 선언은 다음과 같다.

```
HPEN CreatePen(int style, int width, COLORREF color);
```

style 파라메터는 작성할 펜의 형식을 결정하는 파라메터이다. 이것은 다음의 어느 한 값으로 된다.

마 크 로	펜의 형식
PS_DASH	파선
PS_DASHDOT	한점파선
PS_DASHDOTDOT	두점파선
PS_DOT	점선
PS_INSIDEFRAME	둘러 막힌 령역의 내부에 속하는 실선
PS_NULL	빈 펜
PS_SOLID	실선

점선이나 파선은 너비가 1 인 펜으로만 지정할수 있다. *PS_INSIDEFRAME* 펜은 펜의 너비가 1 보다 큰 경우에도 그린 내용이 객체안에 들어간다.

실례로 형식이 PS_INSIDEFRAME 이고 너비가 1 보다 큰 펜을 사용하여 4 각형을 그리면 외형을 그리는 직선이 4 각형령역을 벗어 나지 않게 된다. 그러나 PS_INSIDEFRAME 이 아닌 형식으로 너비가 넓은 펜을 사용하면 직선이 부분적으로 객체 즉 4 각형의 령역을 벗어 나게 된다.

펜의 너비는 론리단위로 width 에 설정한다. 펜의 색은 COLORREF 의 값으로 color 에 설정한다. CreatePen( )은 호출이 성공하면 펜의 손잡이를 돌려 주며 실패하면 NULL 을 돌려 준다.

펜이 작성되면 SelectObject( )를 사용하여 펜을 장치상황에 선택한다. 실례로 아래의 프로그람코드는 붉은펜을 작성하고 그것을 장치상황에 선택한다.

```
HPEN hRedPen;
HRedPen = CreatePen(PS_SOLID, 1, RGB(255,0,0));
SelectObject(dc, hRedpen);
```

작성된 전용펜은 프로그람이 완료하기전에 DeleteObject( )를 사용하여 삭제하여야 한다.

전용붓의 작성

내장펜의 경우와 같이 Windows 2000 이 제공하는 내장붓은 그것을 사용하여 실현할 수 있는것이 제한되여 있으므로 개성적인 붓, 다시말하여 전용붓을 작성하여 리용하는 경우가 많다. 전용붓은 전용펜과 류사한 방법으로 작성된다. 여러가지 형식의 전용붓을

작성할수 있다. 가장 일반적인 전용붓은 *균일붓*(solid brush)이다. 균일붓은 *CreateSolidBrush( )*라는 API 함수를 사용하여 작성된다. 선언은 다음과 같다.

```
HBRUSH CreateSolidBrush(COLORREF color);
```

붓의 색을 color 에 설정하고 함수를 호출하면 붓의 손잡이가 돌려 진다. 호출이 실패하면 NULL 이 돌려 진다.

전용붓이 작성되면 그것을 SelectObject( )를 사용하여 장치상황에 선택한다. 실례로 아래의 프로그람코드는 록색의 붓을 작성하고 그것을 장치상황에 선택한다.

```
HBRUSH hGreenbrush;

hGreenbrush = CreateSolidBrush(RGB(0, 255, 0));
SelectObject(dc, hGreenbrush);
```

작성할수 있는 전용붓의 형식에는 균일붓외에도 두가지 종류가 더 있다. 그것은 *무늬붓*(Pattern brush)과 *줄무늬붓*(Hatch brush)이다. 무늬붓은 비트매프의 무늬로 령역을 색칠하는 붓이다. 줄무늬붓은 몇가지 줄무늬모양을 사용한다. 이 붓들은 *CreatePattern Brush( )* 및 *CreateHatchBrush( )*를 사용하여 작성된다. 선언은 다음과 같다.

```
HBRUSH CreatePatternBrush(HBITMAP hBMap);
HBRUSH CreateHatchBrush(int Style, COLORREF color);
```

*CreatePatternBrush()*에서는 붓의 무늬로 사용되는 비트매프의 손잡이를 hBMap 에 설정한다. 이 비트매프가 령역이나 배경의 색칠에 사용된다. 이 함수는 붓의 손잡이를 돌려 주며 오유가 발생한 경우에는 NULL 을 돌려 준다.

***참고** : 장치독립비트매프를 사용하여 붓을 작성하는 경우에는 CreateDIBPatternBrushPt( )를 사용한다.*

*CreateHatchBrush( )*를 사용하여 작성된 붓은 줄무늬모양으로 된다. Style 에 설정되는 줄무늬모양의 형식으로는 다음의 어느 한 값을 설정한다.

형　　식	줄무늬모양
HS_BDIAGONAL	오른쪽 아래로 향한 경사선

HS_CROSS	격자모양
HS_DIAGCROSS	경사진 격자모양
HS_FDIAGONAL	오른쪽 우로 향한 경사선
HS_HORIZONTAL	수평
HS_VERTICAL	수직

붓의 색은 COLORREF 의 값으로 color 에 설정된다. 이 함수는 붓의 손잡이를 돌려 주며 호출이 실패한 경우에는 NULL 을 돌려 준다.

전용펜과 전용붓의 삭제

필요 없게 된 전용펜과 전용붓은 삭제하여야 한다. 이것은 DeleteObject( )라는 API 함수를 사용하여 진행된다. 내장객체는 삭제할수 없다.(엄밀히 말하면 삭제해서는 안된다.) 객체가 어떤 장치상황에 선택된 상태에서는 삭제할수 없다.

출력방식의 설정

프로그람에서 창문에 도형을 출력할 때는 그때 설정되여 있는 출력방식에 의해 출력을 창문에 복사하는 방법이 결정된다. 체계설정으로는 출력이 창문에 그대로 복사되여 현재의 내용을 덧쓰기하지만 다른 출력방식을 사용할수도 있다.

실례로 출력과 창문의 현재 내용을 AND 연산, OR 연산 및 XOR 연산할수도 있다. *출력방식*을 설정하기 위해서는 *SetROP2( )*함수를 사용한다. 선언은 다음과 같다.

```
int SetROP2(HDC hdc, int Mode);
```

hdc 에는 대상으로 되는 장치상황의 손잡이를 설정하고 Mode 에는 새로운 출력방식을 설정한다. SetROP2( )은 이 함수를 호출하기 전의 출력방식을 돌려 주며 호출이 실패하면 령을 돌려 준다. Mode 에는 표 9-1 에 준 값들가운데서 어느 한 값을 설정한다.

표 9-1. 출력방식의 설정값

마크로	출력방식
R2_BLACK	출력이 검은색으로 된다.
R2_COPYPEN	출력이 창문에 복사되고 현재 화면의 내용에 덧

	쓰기한다.
R2_MASKPEN	출력이 현재 화면의 내용과 AND 연산된다.
R2_MASKNOTPEN	출력을 반전한것과 현재화면의 내용이 AND 연산된다.
R2_MASKPENNOT	출력이 현재화면의 내용을 반전한것과 AND 연산된다.
R2_MERGEPEN	출력이 현재화면의 내용과 OR 연산된다.
R2_MERGENOTPEN	출력을 반전한것과 현재화면의 내용이 OR 연산된다.
R2_MERGEPENNOT	출력이 현재화면의 내용을 반전한것과 OR 연산된다.
R2_NOP	출력이 변경된다.
R2_NOT	출력이 현재화면의 내용을 반전시킨것으로 된다.
R2_NOTCOPYPEN	출력을 반전시킨것이 창문에 복사된다.
R2_NOTMASKPEN	출력이 R2_MASKPEN 을 반전한것으로 된다.
R2_NOTMERGEPEN	출력이 R2_MERGEPEN 을 반전한것으로 된다.
R2_NOTXORPEN	출력이 R2_XORPEN 을 반전한것으로 된다.
R2_WHITE	출력이 흰색으로 된다.
R2_XORPEN	출력이 현재화면의 내용과 XOR 연산된다.

R2_XORPEN 은 화면상의 원래의 정보를 잃지 않고 일시적인 출력을 진행하는 경우에 편리하게 사용된다. 이것은 XOR 연산이 가지는 특수한 기능에 의해 실현된다.

실례로 A 라는 값과 B 라는 값을 XOR 연산하고 다시 한번 B 로 XOR 연산하면 본래의 A 로 돌아간다. 출력과 화면을 XOR 연산하고 다시 한번 동일한 출력으로 XOR 연산하면 화면의 내용은 원래 상태로 복귀된다.

익것은 화면에 어떤것을 일시적으로 표시할 때 XOR 연산하여 출력하면 간단히 다시 한번 XOR 연산하여 화면을 본래의 내용으로 돌려보낼수 있다는것을 의미한다.

이 기술은 제 7 장에서 비트매프의 출력과 관련한 설명을 할 때도 간단히 설명되였다. 다음에 작성하는 도형의 실례프로그람에서도 이 기술을 리용한다.

겸하여 알아둘것은 ROP 는 Raster Operation 의 략어라는것이다. 이 기능은 화면표시장치같은 주사선장치에만 제공된다.

도형의 실례

실례 9-1 의 프로그람은 지금까지 설명한 여러가지 도형함수들의 실례를 보여 주는

것이다. 이 프로그람은 기능이 제한된 간단한 그림그리기(paint)프로그람이다. 이 프로그람에서는 WM_PAINT 통보문을 받아 들일 때 7 장에서 설명한 가상창문기술을 리용하여 창문을 다시그리기한다.

실례 9-1. Graph 프로그람

```
// 간단한 그림그리기프로그람

#include <windows.h>
#include "graph.h"

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);

char szWinName[] = "MyWin"; // 창문클라스의 이름

int maxX, maxY; // 화면의 크기

HDC memdc; // 기억기장치상황의 손잡이
HBITMAP hbit; // 호환성 있는 비트매프의 손잡이
HBRUSH hCurrentbrush, hOldbrush; // 붓의 손잡이
HBRUSH hRedbrush, hGreenbrush, hBluebrush, hNullbrush;

// 펜의 작성
HPEN hOldpen; // 펜의 손잡이
HPEN hCurrentpen; // 현재 선택되여 있는 펜
HPEN hRedpen, hGreenpen, hBluepen, hBlackpen;

int X=0, Y=0;
int pendown = 0;
int endpoints = 0;
int StartX=0, StartY=0, EndX=0, EndY=0;
int Mode;

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HWND hwnd;
```

```
MSG msg;
WNDCLASSEX wcl;
HACCEL hAccel;

// 창문클라스를 정의한다.
wcl.cbSize = sizeof(WNDCLASSEX);

wcl.hInstance = hThisInst;      // 실체의 손잡이
wcl.lpszClassName = szWinName; // 창문클라스의 이름
wcl.lpfnWndProc = WindowFunc;  // 창문함수
wcl.style = 0;                  // 체계설정의 형식

wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

wcl.lpszMenuName = "GraphMenu"; // 기본차림표

wcl.cbClsExtra = 0; // 보조기억기령역은 필요 없다.
wcl.cbWndExtra = 0;

// 창문의 배경색을 흰색으로 한다.
wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

// 창문클라스를 등록한다.
if(!RegisterClassEx(&wcl)) return 0;

/* 창문클라스가 등록되였으므로
   창문을 작성할수 있다. */
hwnd = CreateWindow(
  szWinName, // 창문클라스의 이름
  "A Simple Paint Program", // 제목
  WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
  CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 너비는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 높이는 Windows가 결정하게 한다.
  NULL,          // 어미창문은 없다.
```

```
    NULL,          // 차림표는 없다.
    hThisInst,     // 실체의 손잡이
    NULL           // 추가파라메터는 없다.
  );

  // 건반가속기를 적재한다.
  hAccel = LoadAccelerators(hThisInst, "GraphMenu");

  // 창문을 표시한다.
  ShowWindow(hwnd, nWinMode);
  UpdateWindow(hwnd);

  // 통보문순환고리를 작성한다.
  while(GetMessage(&msg, NULL, 0, 0))
  {
    if(!TranslateAccelerator(hwnd, hAccel, &msg)) {
      TranslateMessage(&msg); // 건반통보를 변환한다.
      DispatchMessage(&msg); // Windows 2000 에 조종을 넘긴다.
    }
  }

  return msg.wParam;
}

/* 이 함수는 Windows 2000 으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받아 들인다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  HDC hdc;
  PAINTSTRUCT paintstruct;
  int response;

  switch(message) {
    case WM_CREATE:
      // 화면의 크기를 얻는다.
      maxX = GetSystemMetrics(SM_CXSCREEN);
```

```
        maxY = GetSystemMetrics(SM_CYSCREEN);

        // 가상창문을 작성한다.
        hdc = GetDC(hwnd);
        memdc = CreateCompatibleDC(hdc);
        hbit = CreateCompatibleBitmap(hdc, maxX, maxY);
        SelectObject(memdc, hbit);
        hCurrentbrush = (HBRUSH) GetStockObject(WHITE_BRUSH);
        SelectObject(memdc, hCurrentbrush);
        PatBlt(memdc, 0, 0, maxX, maxY, PATCOPY);

        // 펜을 작성한다.
        hRedpen = CreatePen(PS_SOLID, 1, RGB(255,0,0));
        hGreenpen = CreatePen(PS_SOLID, 1, RGB(0,255,0));
        hBluepen = CreatePen(PS_SOLID, 1, RGB(0,0,255));

        // 붓을 작성한다.
        hRedbrush = CreateSolidBrush(RGB(255,0,0));
        hGreenbrush = CreateSolidBrush(RGB(0,255,0));
        hBluebrush = CreateSolidBrush(RGB(0,0,255));
        hNullbrush = (HBRUSH) GetStockObject(HOLLOW_BRUSH);

        // 체계설정의 펜을 보관한다.
        hBlackpen = hOldpen = (HPEN) SelectObject(memdc, hRedpen);
        hCurrentpen = hOldpen;
        SelectObject(memdc, hOldpen);

        ReleaseDC(hwnd, hdc);
        break;
    case WM_RBUTTONDOWN: // 령역의 정의를 개시한다.
        endpoints = 1;
        X = StartX = LOWORD(lParam);
        Y = StartY = HIWORD(lParam);
        break;
    case WM_RBUTTONUP: // 령역의 정의를 완료한다.
        endpoints = 0;
        EndX = LOWORD(lParam);
        EndY = HIWORD(lParam);
```

```
      break;
    case WM_LBUTTONDOWN: // 그리기를 개시한다.
      pendown = 1;
      X = LOWORD(lParam);
      Y = HIWORD(lParam);
      break;
    case WM_LBUTTONUP: // 그리기를 완료한다.
      pendown = 0;
      break;
    case WM_MOUSEMOVE:
      if(pendown) { // 그리기한다.
        hdc = GetDC(hwnd);
        SelectObject(memdc, hCurrentpen);
        SelectObject(hdc, hCurrentpen);
        MoveToEx(memdc, X, Y, NULL);
        MoveToEx(hdc, X, Y, NULL);
        X = LOWORD(lParam);
        Y = HIWORD(lParam);
        LineTo(memdc, X, Y);
        LineTo(hdc, X, Y);
        ReleaseDC(hwnd, hdc);
      }
      if(endpoints) { // 령역을 둘러싸는 선을 표시한다.
        hdc = GetDC(hwnd);

        // 점선을 그리는 펜을 선택한다.
        hOldpen = (HPEN) SelectObject(hdc, hRedpen);

        // 출력방식을 XOR 로 변경한다.
        Mode = SetROP2(hdc, R2_XORPEN);

        // 그리기를 하나 칠하지는 않는다.
        hOldbrush =
            (HBRUSH) SelectObject(hdc, GetStockObject(HOLLOW_BRUSH));

        // 낡은 점선 4각형을 소거한다.
        Rectangle(hdc, StartX, StartY, X, Y);
```

```
      X = LOWORD(lParam);
      Y = HIWORD(lParam);

      // 새로운 점선 4각형을 그린다.
      Rectangle(hdc, StartX, StartY, X, Y);

      // 체계설정의 붓으로 복귀한다.
      SelectObject(hdc, hOldbrush);
      SelectObject(hdc, hOldpen);
      SetROP2(hdc, Mode);
      ReleaseDC(hwnd, hdc);
    }
    break;

  case WM_COMMAND:
    switch(LOWORD(wParam)) {
      case IDM_LINE:
        // 현재의 펜을 선택한다.
        SelectObject(memdc, hCurrentpen);

        // 직선을 긋는다.
        MoveToEx(memdc, StartX, StartY, NULL);
        LineTo(memdc, EndX, EndY);

        InvalidateRect(hwnd, NULL, 1);
        break;
      case IDM_RECTANGLE:
        // 붓과 펜을 선택한다.
        SelectObject(memdc, hCurrentbrush);
        SelectObject(memdc, hCurrentpen);

        // 4각형을 그린다.
        Rectangle(memdc, StartX, StartY, EndX, EndY);

        InvalidateRect(hwnd, NULL, 1);
        break;
      case IDM_ELLIPSE:
        // 붓과 펜을 선택한다.
```

```
        SelectObject(memdc, hCurrentbrush);
        SelectObject(memdc, hCurrentpen);

        // 타원을 그린다.
        Ellipse(memdc,  StartX, StartY, EndX, EndY);

        InvalidateRect(hwnd, NULL, 1);
        break;
      case IDM_RED:
        hCurrentpen = hRedpen;
        break;
      case IDM_BLUE:
        hCurrentpen = hBluepen;
        break;
      case IDM_GREEN:
        hCurrentpen = hGreenpen;
        break;
      case IDM_BLACK:
        hCurrentpen = hBlackpen;
        break;
      case IDM_REDFILL:
        hCurrentbrush = hRedbrush;
        break;
      case IDM_BLUEFILL:
        hCurrentbrush = hBluebrush;
        break;
      case IDM_GREENFILL:
        hCurrentbrush = hGreenbrush;
        break;
      case IDM_WHITEFILL:
        hCurrentbrush = hOldbrush;
        break;
      case IDM_NULLFILL:
        hCurrentbrush = hNullbrush;
        break;
      case IDM_RESET:
        // 현재위치를 (0, 0)으로 재설정한다.
        MoveToEx(memdc, 0, 0, NULL);
```

```
            // 배경을 칠하여 소거한다.
            SelectObject(memdc, hOldbrush);
            PatBlt(memdc, 0, 0, maxX, maxY, PATCOPY);

            InvalidateRect(hwnd, NULL, 1);
            break;
          case IDM_EXIT:
            response = MessageBox(hwnd, "Quit the Program?",
                                  "Exit", MB_YESNO);
            if(response == IDYES) PostQuitMessage(0);
            break;
          case IDM_HELP:
            MessageBox(hwnd, "F2: Line\nF3: Rectangle\n"
                        "F4: Ellipse\nF5: Reset",
                        "Paint Hot Keys", MB_OK);
            break;
        }
        break;
      case WM_PAINT: // 다시그리기요구를 처리한다.
        hdc = BeginPaint(hwnd, &paintstruct); // 장치상황을 얻는다.

        // 가상창문을 화면에 복사한다.
        BitBlt(hdc, 0, 0, maxX, maxY, memdc, 0, 0, SRCCOPY);

        EndPaint(hwnd, &paintstruct); // 장치상황을 해제한다.
        break;
      case WM_DESTROY: // 프로그람을 끝낸다.
        DeleteObject(hRedpen); // 펜을 삭제한다.
        DeleteObject(hGreenpen);
        DeleteObject(hBluepen);
        DeleteObject(hBlackpen);

        DeleteObject(hRedbrush); // 붓을 삭제한다.
        DeleteObject(hGreenbrush);
        DeleteObject(hBluebrush);

        DeleteDC(memdc);
```

```
        DeleteObject(hbit);

        PostQuitMessage(0);
        break;
      default:
        /* 이 switch 문에서 지정된것 이외의 통보문은
           Windows 2000 에 처리를 맡긴다. */
        return DefWindowProc(hwnd, message, wParam, lParam);
    }

    return 0;
  }
```

이 프로그람에 필요한 자원파일은 다음과 같다.

```
#include <windows.h>
#include "graph.h"

GraphMenu MENU
{
  POPUP "&Shapes"
  {
    MENUITEM "&Line\F2", IDM_LINE
    MENUITEM "&Rectangle\tF3", IDM_RECTANGLE
    MENUITEM "&Ellipse\tF4", IDM_ELLIPSE
    MENUITEM "E&xit\tCtrl+X", IDM_EXIT
  }
  POPUP "&Options"
  {
    POPUP "&Pen Color"
    {
      MENUITEM "&Red", IDM_RED
      MENUITEM "&Blue", IDM_BLUE
      MENUITEM "&Green", IDM_GREEN
      MENUITEM "Bl&ack", IDM_BLACK
    }
    POPUP "&Fill Color"
```

```
        {
          MENUITEM "&Red", IDM_REDFILL
          MENUITEM "&Blue", IDM_BLUEFILL
          MENUITEM "&Green", IDM_GREENFILL
          MENUITEM "&White", IDM_WHITEFILL
          MENUITEM "&Null", IDM_NULLFILL
        }
        MENUITEM "&Reset\tF5", IDM_RESET
    }
    MENUITEM "&Help", IDM_HELP
}

GraphMenu ACCELERATORS
{
    VK_F1, IDM_HELP, VIRTKEY
    VK_F2, IDM_LINE, VIRTKEY
    VK_F3, IDM_RECTANGLE, VIRTKEY
    VK_F4, IDM_ELLIPSE, VIRTKEY
    VK_F5, IDM_RESET, VIRTKEY
    "^X", IDM_EXIT
}
```

아래의 머리부파일 GRAPH.H도 필요된다.

```
#define IDM_LINE            100
#define IDM_RECTANGLE       101
#define IDM_ELLIPSE         102
#define IDM_RED             103
#define IDM_GREEN           104
#define IDM_BLUE            105
#define IDM_BLACK           106
#define IDM_REDFILL         107
#define IDM_GREENFILL       108
#define IDM_BLUEFILL        109
#define IDM_WHITEFILL       110
#define IDM_NULLFILL        111
#define IDM_RESET           112
#define IDM_HELP            113
```

```
#define IDM_EXIT                          114
```

그리기프로그람의 실행결과는 그림 9-2와 같다.

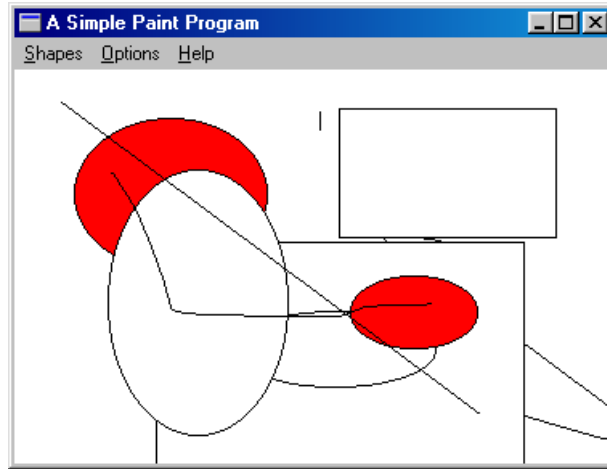



그림 9-2. 그리기프로그람의 실행결과

그리기프로그람은 다음과 같이 동작한다.

마우스를 사용하여 자유로 직선이나 곡선을 그릴수 있다. 그림을 그리려면 왼쪽 단추를 누른채로 마우스를 끌기한다. 오른쪽 단추는 령역을 정의하는데 사용된다. 령역의 시작점에 마우스의 지시자를 이동하고 마우스의 오른쪽 단추를 누른채로 마우스의 지시자를 령역의 종점으로 이동한다. 령역이 목적하는 크기로 되면 마우스의 오른쪽 단추를 놓는다.

령역을 설정하면 령역을 가리키는 4각형이 표시된다. 그리고 정의된 령역의 내부에 직선, 4각형 또는 타원을 그릴수 있다.

이 도형들은 [Shapes]차림표를 선택하여 그릴수 있다. [Options]차림표를 사용하면 새로운 펜이나 붓의 색을 설정하거나 창문의 내용을 소거할수 있다. 그리기프로그람코드의 내용의 대부분은 쉽게 리해할수 있을것이다. 그러나 몇개의 중요한 부분이 있으므로 그것들을 설명하자.

그림그리기프로그람의 상세

그림그리기프로그람을 기동하면 가상창문의 비트매프, 펜, 붓과 같은 몇개의 객체들이 작성된다. 이 객체들은 프로그람이 완료될 때 삭제된다.

변수 pendown의 값은 마우스의 왼쪽 단추가 눌리워 져 있을 때 1로 설정된다. 그밖의 경우에는 령으로 된다. 변수 endpoints 의 값은 마우스의 오른쪽 단추를 누르는것으로서 령역의 정의가 개시되였을 때 1 로 설정된다. 그밖의 경우에는 령으로 된다. 이

변수들은 *WM_MOUSEMOVE* 통보문을 어떻게 처리하는가를 결정하는데 리용된다.

프로그람에서 가장 중요한 부분은 WM_MOUSEMOVE 통보문처리이다. 내용을 주의깊게 살펴 보자. 아래에 다시 한번 프로그람코드를 보여 주었다.

```
case WM_MOUSEMOVE:
    if(pendown) { // 그리기한다.
      hdc = GetDC(hwnd);
      SelectObject(memdc, hCurrentpen);
      SelectObject(hdc, hCurrentpen);
      MoveToEx(memdc, X, Y, NULL);
      MoveToEx(hdc, X, Y, NULL);
      X = LOWORD(lParam);
      Y = HIWORD(lParam);
      LineTo(memdc, X, Y);
      LineTo(hdc, X, Y);
      ReleaseDC(hwnd, hdc);
    }
    if(endpoints) { // 령역을 둘러막는 선을 표시한다.
      hdc = GetDC(hwnd);

      // 점선을 그리는 펜을 선택한다.
      hOldpen = (HPEN) SelectObject(hdc, hRedpen);

      // 출력방식을 XOR로 변경한다.
      Mode = SetROP2(hdc, R2_XORPEN);

      // 그리기는 하지만 칠하지는 않는다.
      hOldbrush =
          (HBRUSH) SelectObject(hdc, GetStockObject(HOLLOW_BRUSH));

      // 낡은 점선 4각형을 소거한다.
      Rectangle(hdc, StartX, StartY, X, Y);

      X = LOWORD(lParam);
      Y = HIWORD(lParam);
```

```
        // 새 점선 4각형을 그린다.
        Rectangle(hdc, StartX, StartY, X, Y);

        // 체계설정의 붓으로 복귀한다.
        SelectObject(hdc, hOldbrush);
        SelectObject(hdc, hOldpen);
        SetROP2(hdc, Mode);
        ReleaseDC(hwnd, hdc);
    }
    break;
```

WM_MOUSEMOVE 통보문은 *마우스*가 움직일 때마다 발송된다. 마우스의 X 자리표는 lParam 의 아래단어에 보관되여 있으며 Y 자리표는 Iparam 의 웃단어에 보관되여 있다.

pendown 이 TRUE 인 경우는 사용자가 직선을 그리고 있다. 이 경우에는 마우스가 움직일 때마다 먼저 현재의 펜이 창문의 장치상황(hdc) 및 가상창문의 장치상황(memdc)에 선택된다.

다음 마우스의 마지막 위치와 새로운 위치를 련결하는 직선이 두 장치상황에 그려진다. 이렇게 되여 마우스를 움직이면 원활하고 틘 부분이 없는 직선을 련속적으로 그리는 기능이 실현된다.(시험적으로 LineTo( )가 아니라 SetPixel( )을 사용하여 결과를 확인해 보라. WM_MOUSEMOVE 는 마우스를 움직이는 자리길의 모든 점에서 생성되지는 않으므로 불련속적인 선으로 되여 버리고 마는것을 보게 될것이다.)

직선이 그려 지면 도형의 현재위치가 현재마우스의 위치로 변경된다. 그것은 LineTo( )가 현재위치를 직선이 끝나는 점으로 변경하기때문이다.

endpoint 가 TRUE 인 경우는 마우스가 4 각형 령역을 정의하는데 사용되고 있다. 첫번째 정점은 WM_RBUTTONDOWN 통보문이 발송될 때 설정된다. 두번째 정점은 WM_RBUTTONUP 통보문이 발송될 때 설정된다. 통보문의 LOWORD(lParam) 및 HIWORD(lParam)에는 각각 마우스의 X, Y 자리표가 보관되여 있다.

마우스를 움직이면 현재령역의 크기를 보여 주는 4 각형이 표시된다. 이 4 각형은 창문과 XOR 연산된다. 이렇게 하는 리유는 XOR 연산을 두번 진행하면 본래의 상태로 복귀되기때문이다. 출력방식으로 *X2_XORPEN* 을 사용한 첫번째 Rectangle( )의 호출이 4 각형을 표시하고 그것을 두번째 호출이 소거한다. 이렇게 되여 창문의 현재내용을 잃지 않고 4 각형을 그릴수 있다.

다시 한보 전진

불규칙도형(Illegular graphics)의 그리기

Windows 2000 은 직선, 타원 및 4 각형 등의 기초도형만이 아니라 불규칙적인 도형을 그리는 기능도 제공하고 있다. 이러한 객체의 하나로서 다각형이 있다. Windows 2000 에 있어서 다각형이란 세개이상의 변을 가지는 닫긴 령역이다. 다각형을 그리자면 *Polygon( )*함수를 사용한다. 선언은 다음과 같다.

```
BOOL Polygon(HDC hdc, CONST POINT *vertices, int num);
```

hdc 에는 장치상황을 설정한다. 다각형의 매 정점을 보관하는 배렬을 vertices 에 설정하고 정점의 수를 num 에 설정한다. 이 함수는 호출이 성공하면 령이 아닌 값을 돌려 주며 실패하면 령을 돌려 준다.

다각형은 현재 선택된 펜으로 그려 지며 현재 선택되여 있는 붓으로 내부가 칠해 진다. Polygon( )함수는 vertices 배렬의 마지막 점과 첫점을 련결하는 직선을 자동적으로 그리므로 닫긴도형으로 된다. Polygon( )은 현재위치를 사용하지 않으며 그 값을 갱신하지도 않는다. Polygon( )을 사용해 보려면 우선 다음과 같은 대역선언을 그리기프로그람에 추가해야 한다.

```
POINT polygon[5] = {
    10, 10,
    10, 40,
    40, 70,
    40, 90,
    10, 10 };
```

다음으로 IDD_LINE 의 case 문에 아래의 프로그람코드를 추가한다.

```
Polygon(memdc, polygon, 5);
```

이렇게 하면 [Line]차림표를 선택할 때마다 5 각형이 표시되게 된다.

API 함수에는 이밖에도 복잡한 객체를 그리기 위한 API 함수들이 여러 개 있다. 대표적인 것은 PolyLine( ), PolyLineTo( ), PolyPolyLine( ), PolyPolygon( ), PolyBezier( ) 및 PolyBezierTo( ) 등이다.

세계변환의 사용법

모든 판본의 Windows 는 그림그리기프로그람에서 활용된 도형기능들을 제공하고 있다. 그러나 Windows 2000 에는 Windows 95/98 에서는 지원되지 않는 강력한 확장기능 즉 *세계변환*이 있다. 세계변환이란 출력과 창문을 대응시키는 방법을 다양하게 변화시키는 변환기능을 실현하는것이다. 변환의 종류에는 회전, 이동, 축척, 자름 및 반전등이 있다. Windows 2000 을 사용하면 일반적인 CAD(Computer Aided Design)에서 볼수 있는 복잡한 도형조작을 손 쉽게 실현할수 있다. 실례로 화상을 회전시키거나 확대할수 있다.

Windows 2000 은 한 자리표공간을 다른 자리표공간으로 넘기기하여 세계변환을 실현한다. 여기서는 Windows 2000 의 자리표공간에 대한 설명으로부터 시작한다.

자리표공간

Windows 2000 은 세가지 *자리표공간*을 정의하고 있다. 첫번째 자리표공간을 *세계자리표공*간이라고 하며 이 자리표공간은 세계변환에서 사용된다. 두번째 자리표공간은 페지공간으로서 이것은 프로그람에서 사용되는 론리자리표공간이다. 세번째 자리표공간은 물리적자리표를 사용하는 *장치공간*으로서 물리적장치자체에 대응된다. 물리적장치를 네번째 자리표공간이라고 부르기도 한다.

알고 있는바와 같이 출력할 때 자리표는 론리단위로 지정된다. 체계설정의 단위는 화소이다. 그외에도 여러가지 단위가 있다. 이러한 단위들은 동시에 한개의 자리표공간에서 한 물리적장치에 넘기기된다. 세계변환기능을 사용하지 않는 경우에는 출력의 넘기기가 페지공간에서부터 개시된다. 세계변환을 사용하는 경우에는 출력의 넘기기가 세계공간에서 시작된다. 자리표공간이 바뀔 때는 앞의 자리표공간의 내용이 뒤의 자리표공간으로 넘기기된다.

례하면 *페지공간*으로부터 장치공간에로의 넘기기에서는 론리단위가 물리단위로 변환된다. 세계변환을 사용할 때는 세계공간으로부터 페지공간에로의 넘기기가 미리 설정된 변환방법(회전,축척 등)으로 된다.

SetWorldTransform( )

세계변환의 변환방법은 *SetWorldTransform( )*에서 설정된다. 선언은 다음과 같다.

```
BOOL SetWorldTransform(HDC hdc, CONST XFORM *lpTransform);
```

hdc 는 변환을 진행하는 장치상황의 손잡이이다. 변환방법이 lpTransform 이 가리키는 구조체로 설정된다. 이 함수는 호출이 성공하면 령이 아닌 값을 돌려 주며 실패하

면 령을 돌려 준다. SetWorldTransform( )을 호출한 다음에는 지정된 변환이 적용된다.
변환방법은 *XFORM* 구조체에서 지정된다. 정의는 다음과 같다.

```
typedef struct tagXFORM
{
    FLOAT eM11;
    FLOAT eM12;
    FLOAT eM21;
    FLOAT em22;
    FLOAT eDx;
    FLOAT eDy;
}   XFORM;
```

기본적인 변환방법은 XFORM 에서 정의된 행렬의 값에 의해 결정된다. 매개 변환이 어떻게 실현되는가를 설명해 보자.

출력을 이동(원점을 이동)시키려면 X 축방향의 이동량을 eDx 에 설정하고 Y 축방향의 이동량을 eDy 에 설정한다. 이동은 설정할수 있는 변환가운데서 가장 간단한 변환이다. 이동에 의해 출력위치가 옮겨 진다. 이동을 다른 변환과 조합시킬수도 있다.

각도 theta 만큼 출력을 회전시키기 위해서는 eM11,eM12,eM21 및 eM22 을 아래의 값으로 설정한다.

성　　원	값
eM11	cos(theta)
eM12	sin(theta)
eM21	-sin(theta)
eM22	cos(theta)

출력을 축척하자면(확대하거나 축소하자면) X 축방향의 축척률을 eM11 에 설정하고 Y 축방향의 축척률을 eM22 에 설정한다. eM12 와 eM21 은 사용하지 않는다.(령을 설정한다.)

출력을 자르기하자면(X 자리표와 Y 자리표를 바꾸어 축척하자면 X 축방향의 자름률을 eM12 에 설정하고 Y 방향의 자름률을 eM21 에 설정한다. eM11 과 eM22 은 사용하지 않는다.(령을 설정한다.)

X 축에 대해 출력을 반전시키기 위해서는 eM11 에 -1 을 설정하고 eM22 에 1 을 설정한다. Y 축에 대해 출력을 반전시키기 위해서는 eM22 에 -1 을 설정하고 eM11 에 1 을 설정한다. 두 축에 대해서 출력을 반전시키기 위해서는 eM11 과 eM22 에 -1 을 설정한다. eM12 과 eM21 은 사용하지 않는다.(령을 설정한다.)

XFORM 구조체의 성원에 여러가지 값을 설정하면 매우 다양한 변환을 할수 있다. 매개 변환의 효과를 리해하기 위한 최량의 방법은 실제로 시험해 보는것이다.

도형방식의 설정

SetWorldTransform( )의 기능을 사용하자면 *SetGraphicsMode( )*함수를 사용하여 사전에 Windows 2000 의 확장도형방식을 유효하게 해야 한다. 선언은 다음과 같다.

int SetGraphicsMode(HDC hdc, int GraphMode);

hdc 에는 대상으로 되는 장치상황의 손잡이를 설정하고 GraphMode 에 목적하는 *도형방식*을 설정한다. 이 값은 *GM_COMPATIBLE* 혹은 *GM_ADVANCE*의 어느 한 값이여야 한다. 이 함수는 바로 전에 설정되여 있던 방식을 돌려 주며 호출이 실패하면 령을 돌려 준다.

체계설정의 도형방식은 *GM_COMPATIBLE* 로 된다. 이것은 Windows 95/98 에서도 사용되는 방식이다. SetWorldTransform( )을 사용하려면 도형방식을 GM_ADVANCE 에 설정하여야 한다.

그림그리기프로그람에 회전기능을 추가

SetWorldTransform( )의 사용방법을 리해하기 위해 그림그리기프로그람의 기능을 확장하여 그린 화상을 회전할수 있게 해 보자.

왼쪽화살건을 누를 때마다 현재 표시되여 있는 화상이 시계바늘회전방향의 반대방향으로 30° 씩 회전한다. 오른쪽 화살건을 누를 때마다 화상이 시계바늘회전방향으로 30° 씩 회전한다. 어느 경우에나 의뢰자구역의 중심을 회전중심으로 하여 화상이 회전한다. 프로그람코드를 실례 9-2 에 보여 주었다. 프로그람의 실행결과는 그림 9-3 에 보여 주었다.

실례 9-2. Rotation 프로그람

```
// SetWorldTransform( )을 리용한 화상의 회전

#include <windows.h>
#include "graph.h"
#include <cmath>

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);

char szWinName[] = "MyWin"; // 창문클라스의 이름

int maxX, maxY; // 화면의 크기
```

```
HDC memdc; // 기억기장치상황의 손잡이
HBITMAP hbit; // 호환성 있는 비트매프의 손잡이
HBRUSH hCurrentbrush, hOldbrush; // 붓의 손잡이
HBRUSH hRedbrush, hGreenbrush, hBluebrush, hNullbrush;

// 펜의 작성
HPEN hOldpen; // 펜의 손잡이
HPEN hCurrentpen; // 현재 선택되여 있는 펜
HPEN hRedpen, hGreenpen, hBluepen, hBlackpen;

int X=0, Y=0;
int pendown = 0;
int endpoints = 0;
int StartX=0, StartY=0, EndX=0, EndY=0;
int Mode;

int theta = 0; // 회전각도

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;
  HACCEL hAccel;

  // 창문클라스를 정의한다.
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst;      // 실체의 손잡이
  wcl.lpszClassName = szWinName; // 창문클라스의 이름
  wcl.lpfnWndProc = WindowFunc;  // 창문함수
  wcl.style = 0;                 // 체계설정의 형식

  wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
  wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
  wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식
```

```
wcl.lpszMenuName = "GraphMenu"; // 기본차림표

wcl.cbClsExtra = 0; // 보조기억기령역은 필요 없다.
wcl.cbWndExtra = 0;

// 창문의 배경색을 흰색으로 한다.
wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

// 창문클라스를 등록한다.
if(!RegisterClassEx(&wcl)) return 0;

/* 창문클라스가 등록되였으므로
   창문을 작성할수 있다. */
hwnd = CreateWindow(
  szWinName, // 창문클라스의 이름
  "Paint With Rotation", // 제목
  WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
  CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 너비는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 높이는 Windows가 결정하게 한다.
  NULL,          // 어미창문은 없다.
  NULL,          // 차림표는 없다.
  hThisInst,     // 실체의 손잡이
  NULL           // 추가파라메터는 없다.
);

// 건반가속기를 적재한다.
hAccel = LoadAccelerators(hThisInst, "GraphMenu");

// 창문을 표시한다.
ShowWindow(hwnd, nWinMode);
UpdateWindow(hwnd);

// 통보문순환고리를 작성한다.
while(GetMessage(&msg, NULL, 0, 0))
{
```

```
        if(!TranslateAccelerator(hwnd, hAccel, &msg)) {
          TranslateMessage(&msg); // 건반통보를 변환한다.
          DispatchMessage(&msg); // Windows 2000 에 조종을 넘긴다.
        }
      }

    return msg.wParam;
  }

/* 이 함수는 Windows 2000 으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받아 들인다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  HDC hdc;
  PAINTSTRUCT paintstruct;
  int response;
  RECT rect;
  XFORM tf; // 변환행렬

  switch(message) {
    case WM_CREATE:
      // 화면의 크기를 얻는다.
      maxX = GetSystemMetrics(SM_CXSCREEN);
      maxY = GetSystemMetrics(SM_CYSCREEN);

      // 가상창문을 작성한다.
      hdc = GetDC(hwnd);
      memdc = CreateCompatibleDC(hdc);
      hbit = CreateCompatibleBitmap(hdc, maxX, maxY);
      SelectObject(memdc, hbit);
      hCurrentbrush = (HBRUSH) GetStockObject(WHITE_BRUSH);
      SelectObject(memdc, hCurrentbrush);
      PatBlt(memdc, 0, 0, maxX, maxY, PATCOPY);

      // 펜을 작성한다.
      hRedpen = CreatePen(PS_SOLID, 1, RGB(255,0,0));
```

```
        hGreenpen = CreatePen(PS_SOLID, 1, RGB(0,255,0));
        hBluepen = CreatePen(PS_SOLID, 1, RGB(0,0,255));

        // 붓을 작성한다.
        hRedbrush = CreateSolidBrush(RGB(255,0,0));
        hGreenbrush = CreateSolidBrush(RGB(0,255,0));
        hBluebrush = CreateSolidBrush(RGB(0,0,255));
        hNullbrush = (HBRUSH) GetStockObject(HOLLOW_BRUSH);

        // 체계설정의 펜을 보관한다.
        hBlackpen = hOldpen = (HPEN) SelectObject(memdc, hRedpen);
        hCurrentpen = hOldpen;
        SelectObject(memdc, hOldpen);

        ReleaseDC(hwnd, hdc);
        break;
      case WM_KEYDOWN: // 화상을 회전한다.
        switch((char) wParam) {
          case VK_LEFT: // 시계바늘의 반대방향
            theta -= 30;
            InvalidateRect(hwnd, NULL, 1);
            break;
          case VK_RIGHT: // 시계바늘 방향
            theta += 30;
            InvalidateRect(hwnd, NULL, 1);
            break;
        }
        break;
      case WM_RBUTTONDOWN: // 령역의 정의를 개시한다.
        if(theta) { // 화면을 원상태로 복귀한다.
          theta = 0;
          InvalidateRect(hwnd, NULL, 1);
        }
        endpoints = 1;
        X = StartX = LOWORD(lParam);
        Y = StartY = HIWORD(lParam);
        break;
      case WM_RBUTTONUP: // 령역의 정의를 끝낸다.
```

```
        endpoints = 0;
        EndX = LOWORD(lParam);
        EndY = HIWORD(lParam);
        break;
      case WM_LBUTTONDOWN: // 그리기를 개시한다.
        if(theta) { // 화면을 원상태로 복귀한다.
          theta = 0;
          InvalidateRect(hwnd, NULL, 1);
        }
        pendown = 1;
        X = LOWORD(lParam);
        Y = HIWORD(lParam);
        break;
      case WM_LBUTTONUP: // 그리기를 끝낸다.
        pendown = 0;
        break;
      case WM_MOUSEMOVE:
        if(pendown) { // 그린다.
          hdc = GetDC(hwnd);
          SelectObject(memdc, hCurrentpen);
          SelectObject(hdc, hCurrentpen);
          MoveToEx(memdc, X, Y, NULL);
          MoveToEx(hdc, X, Y, NULL);
          X = LOWORD(lParam);
          Y = HIWORD(lParam);
          LineTo(memdc, X, Y);
          LineTo(hdc, X, Y);
          ReleaseDC(hwnd, hdc);
        }
        if(endpoints) { // 영역을 둘러싸는 선을 표시한다.
          hdc = GetDC(hwnd);

          // 점선을 그리는 펜을 선택한다.
          hOldpen = (HPEN) SelectObject(hdc, hRedpen);

          // 출력방식을 XOR로 변경한다.
          Mode = SetROP2(hdc, R2_XORPEN);
```

```
      // 그리기를 하지만 색칠하지는 않는다.
      hOldbrush =
        (HBRUSH) SelectObject(hdc, GetStockObject(HOLLOW_BRUSH));

      // 낡은 점선 4각형을 소거한다.
      Rectangle(hdc, StartX, StartY, X, Y);

      X = LOWORD(lParam);
      Y = HIWORD(lParam);

      // 새로운 점선 4각형을 그린다.
      Rectangle(hdc, StartX, StartY, X, Y);

      // 체계설정의 붓으로 복귀한다.
      SelectObject(hdc, hOldbrush);
      SelectObject(hdc, hOldpen);
      SetROP2(hdc, Mode);
      ReleaseDC(hwnd, hdc);
    }
    break;
  case WM_COMMAND:
    switch(LOWORD(wParam)) {
      case IDM_ROTATERESET:
        theta = 0;
        InvalidateRect(hwnd, NULL, 1);
        break;
      case IDM_LINE:
        // 현재의 펜을 선택한다.
        SelectObject(memdc, hCurrentpen);

        // 직선을 긋는다.
        MoveToEx(memdc, StartX, StartY, NULL);
        LineTo(memdc, EndX, EndY);

        InvalidateRect(hwnd, NULL, 1);
        break;
      case IDM_RECTANGLE:
        // 붓과 펜을 선택한다.
```

```
    SelectObject(memdc, hCurrentbrush);
    SelectObject(memdc, hCurrentpen);

    // 4각형을 그린다.
    Rectangle(memdc, StartX, StartY, EndX, EndY);

    InvalidateRect(hwnd, NULL, 1);
    break;
  case IDM_ELLIPSE:
    // 붓과 펜을 선택한다.
    SelectObject(memdc, hCurrentbrush);
    SelectObject(memdc, hCurrentpen);

    // 타원을 그린다.
    Ellipse(memdc,  StartX, StartY, EndX, EndY);

    InvalidateRect(hwnd, NULL, 1);
    break;
  case IDM_RED:
    hCurrentpen = hRedpen;
    break;
  case IDM_BLUE:
    hCurrentpen = hBluepen;
    break;
  case IDM_GREEN:
    hCurrentpen = hGreenpen;
    break;
  case IDM_BLACK:
    hCurrentpen = hBlackpen;
    break;
  case IDM_REDFILL:
    hCurrentbrush = hRedbrush;
    break;
  case IDM_BLUEFILL:
    hCurrentbrush = hBluebrush;
    break;
  case IDM_GREENFILL:
    hCurrentbrush = hGreenbrush;
```

```
            break;
          case IDM_WHITEFILL:
            hCurrentbrush = hOldbrush;
            break;
          case IDM_NULLFILL:
            hCurrentbrush = hNullbrush;
            break;
          case IDM_RESET:
            // 현재위치를 (0, 0)으로 재설정한다.
            MoveToEx(memdc, 0, 0, NULL);

            // 배경을 칠하여 소거한다.
            SelectObject(memdc, hOldbrush);
            PatBlt(memdc, 0, 0, maxX, maxY, PATCOPY);

            theta = 0;

            InvalidateRect(hwnd, NULL, 1);
            break;
          case IDM_EXIT:
            response = MessageBox(hwnd, "Quit the Program?",
                                  "Exit", MB_YESNO);
            if(response == IDYES) PostQuitMessage(0);
            break;
          case IDM_HELP:
            MessageBox(hwnd, "F2: Line\nF3: Rectangle\n"
                       "F4: Ellipse\nF5: Rotation Reset\n"
                       "F6: Reset\n",
                       "Paint and Rotate", MB_OK);
            break;
        }
        break;
    case WM_PAINT: // 다시그리기요구를 처리한다.
      hdc = BeginPaint(hwnd, &paintstruct); // 장치상황을 얻는다.

      // Windows 2000 확장도형방식을 사용한다.
      SetGraphicsMode(hdc, GM_ADVANCED);
```

```
        // 의뢰자구역의 현재크기를 얻는다.
        GetClientRect(hwnd, &rect);

        // 화상을 회전시킨다.
        tf.eM11 = (float) cos(theta * 3.1416/180);
        tf.eM12 = (float) sin(theta * 3.1416/180);
        tf.eM21 = (float) -sin(theta * 3.1416/180);
        tf.eM22 = (float) cos(theta * 3.1416/180);
        tf.eDx = (float) rect.right/2;
        tf.eDy = (float) rect.bottom/2;
        SetWorldTransform(hdc, &tf); // 변환을 설정한다.

        // 가상창문을 화면에 복사한다.
        BitBlt(hdc, -rect.right/2, -rect.bottom/2,
               rect.right, rect.bottom, memdc, 0, 0, SRCCOPY);

        EndPaint(hwnd, &paintstruct); // 장치상황을 해제한다.
        break;
      case WM_DESTROY: // 프로그람을 끝낸다.
        DeleteObject(hRedpen); // 펜을 삭제한다.
        DeleteObject(hGreenpen);
        DeleteObject(hBluepen);
        DeleteObject(hBlackpen);

        DeleteObject(hRedbrush); // 붓을 삭제한다.
        DeleteObject(hGreenbrush);
        DeleteObject(hBluebrush);

        DeleteDC(memdc);
        DeleteObject(hbit);

        PostQuitMessage(0);
        break;
      default:
        /* 이 switch 문에서 지정된것 이외의 통보문은
           Windows 2000 에 처리를 맡긴다. */
        return DefWindowProc(hwnd, message, wParam, lParam);
    }
```

```
  return 0;
}
```

이 프로그람은 다음의 자원파일을 사용한다.

```
#include <windows.h>
#include "graph.h"

GraphMenu MENU
{
  POPUP "&Shapes"
  {
    MENUITEM "&Line\tF2", IDM_LINE
    MENUITEM "&Rectangle\tF3", IDM_RECTANGLE
    MENUITEM "&Ellipse\tF4", IDM_ELLIPSE
    MENUITEM "E&xit\tCtrl+X", IDM_EXIT
  }
  POPUP "&Options"
  {
    POPUP "&Pen Color"
    {
      MENUITEM "&Red", IDM_RED
      MENUITEM "&Blue", IDM_BLUE
      MENUITEM "&Green", IDM_GREEN
      MENUITEM "Bl&ack", IDM_BLACK
    }
    POPUP "&Fill Color"
    {
      MENUITEM "&Red", IDM_REDFILL
      MENUITEM "&Blue", IDM_BLUEFILL
      MENUITEM "&Green", IDM_GREENFILL
      MENUITEM "&White", IDM_WHITEFILL
      MENUITEM "&Null", IDM_NULLFILL
    }
    MENUITEM "Ro&tation Reset\tF5", IDM_ROTATERESET
    MENUITEM "&Reset\tF6", IDM_RESET
  }
```

```
    MENUITEM "&Help", IDM_HELP
}

GraphMenu ACCELERATORS
{
  VK_F1, IDM_HELP, VIRTKEY
  VK_F2, IDM_LINE, VIRTKEY
  VK_F3, IDM_RECTANGLE, VIRTKEY
  VK_F4, IDM_ELLIPSE, VIRTKEY
  VK_F5, IDM_ROTATERESET, VIRTKEY
  VK_F6, IDM_RESET, VIRTKEY
  "^X", IDM_EXIT
}
```

GRAPH.H 에는 다음 행을 추가한다.

```
#define IDM_ROTATERESET    200
```

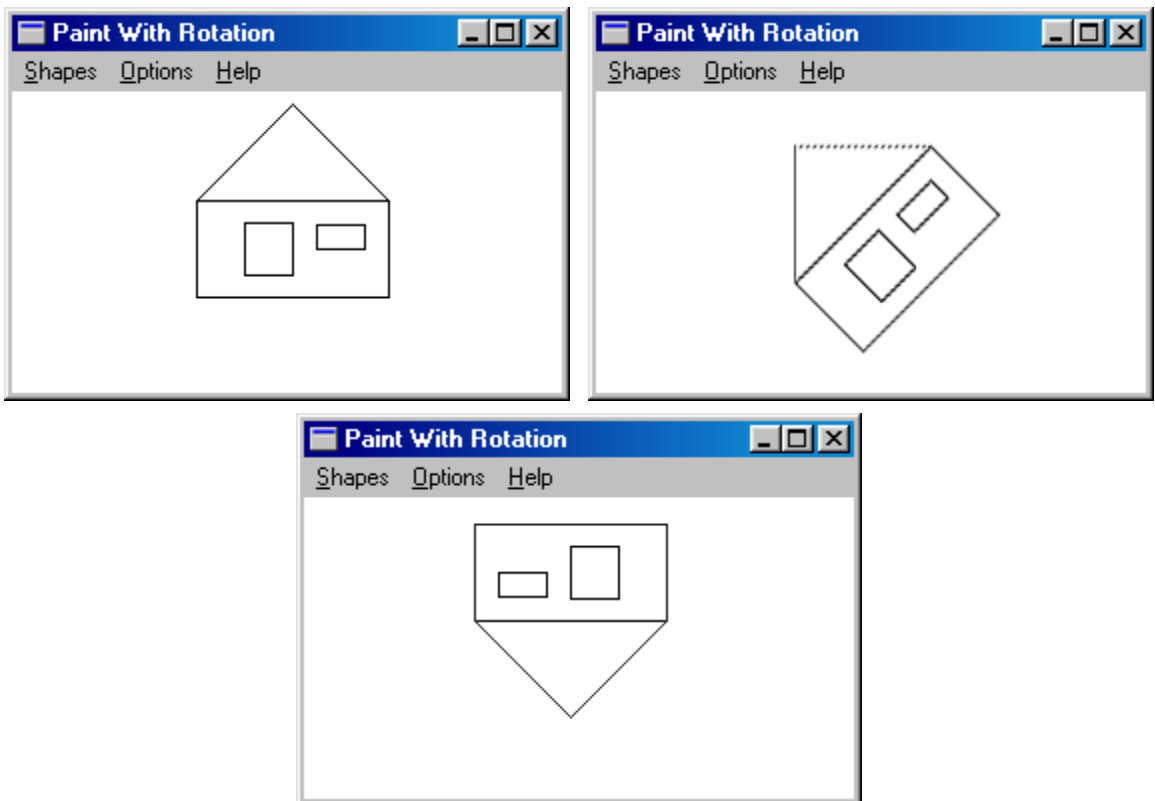



그림 9-3. 회전프로그람의 실행결과

화상을 회전시키는 절차

화상을 회전시키기 위해 그림그리기프로그람에 다음의 내용을 추가한다.

- 도단위로 현재회전각도를 보관하는 대역변수 theta
- 현재의 변환행렬을 보관하는 국부변수 tf
- 건이 눌리울 때마다 30° 씩 theta를 증감하는 왼쪽 화살건과 오른쪽 화살건의 처리
- WM_PAINT 에서 SetWorldTransform( )의 호출
- [Rotation Reset]차림표

이것들이 어떻게 작용하는가를 설명한다. theta 의 값은 WM_PAINT 가운데서 가상창문(memdc)으로부터 물리장치상황(hdc)에 화상이 복사될 때 적용되는 각도를 결정한다. VK_LEFT 및 VK_RIGHT 의 처리에서는 건이 눌리울 때마다 theta 의 값을 증가하거나 감소시키고 창문이 다시그리기된다.

실제로 화상을 회전하는 처리는 아래에 준 WM_PAINT 안에서 진행된다. 프로그람코드의 내용을 상세하게 조사해 보자.

```
case WM_PAINT: // 다시그리기요구를 처리한다.
  hdc = BeginPaint(hwnd, &paintstruct); // 장치상황을 얻는다.

  // Windows 2000 확장도형방식을 사용한다.
  SetGraphicsMode(hdc, GM_ADVANCED);

  // 의뢰자구역의 현재크기를 얻는다.
  GetClientRect(hwnd, &rect);

  // 화상을 회전시킨다.
  tf.eM11 = (float) cos(theta * 3.1416/180);
  tf.eM12 = (float) sin(theta * 3.1416/180);
  tf.eM21 = (float) -sin(theta * 3.1416/180);
  tf.eM22 = (float) cos(theta * 3.1416/180);
  tf.eDx = (float) rect.right/2;
  tf.eDy = (float) rect.bottom/2;
  SetWorldTransform(hdc, &tf); // 변환을 설정한다.
```

```
        // 가상창문을 화면에 복사한다.
        BitBlt(hdc, -rect.right/2, -rect.bottom/2,
               rect.right, rect.bottom, memdc, 0, 0, SRCCOPY);

        EndPaint(hwnd, &paintstruct); // 장치상황을 해제한다.
        break;
```

WM_PAINT 통보문을 받아 들일 때마다 장치상황이 얻어지며 확장도형방식이 설정된다. 의뢰자구역의 현재크기가 얻어지고 theta 에 보관된 각도로 회전을 진행하기 위한 변환행렬이 설정된다.

C/C++표준함수로 시누스나 코시누스값을 구하는 경우에는 각도를 rad 단위로 설정한다. 그래서 함수의 파라메터에는 theta 를 ° 단위로부터 rad 단위에로 변환하는 수식이 들어 있다. theta 가 령(이것이 초기값이다.)인 때는 회전을 진행하지 않는다.

변환행렬에서 eDx 에 rect.right/2 를 설정하고 eDy 에 rect.bottom/2 를 설정한데 주목해야 한다. 모든 변환은 원점을 중심으로 진행된다. 원점은 체계설정에서는 창문의 왼쪽 웃모서리로 되여 있다. 이것을 각각 창문의 의뢰자구역의 너비와 높이의 1/2 로 하여 변환의 원점이 창문의 중심으로 된다. 이렇게 되여 창문중심주위로의 회전이 진행되게 된다.

SetWorldTransform( )이 호출되면 지정된 변환이 유효하게 되며 그후의 출력에 적용된다. WM_PAINT 내에서 가상창문의 내용을 물리창문에 복사할 때 *BitBlt( )*를 호출하면 회전변환이 자동적으로 진행된다.

*BitBlt( )*를 호출하는 부분을 잘 살펴 보자. 복사측의 왼쪽웃모서리를 지정하는 값이 (0, 0)이 아니라 –rect.right/2 와 –rect.bottom/2 로 되여 있다. 이것은 변환의 원점이 창문크기의 절반만큼 이동하였기때문이다. 가상창문으로부터 복사할 때 이 보정이 필요하게 된다.

처음의 회전되지 않은 상태로 화상을 복귀하려면 [Option]차림표로부터 [Rotation Reset]를 선택한다. 이 차림표항목은 theta 에 령을 설정한 다음 창문을 다시그리기한다. 이 처리는 마우스의 왼쪽 혹은 오른쪽 단추를 누를 때 진행된다.

자체로 해보기

그림그리기프로그람에 다음의 확장기능을 추가해 보라. 일반 4 각형과 모서리가 원활한 4 각형을 선택하는 기능, 부채형을 그리는 기능, 창문을 부분적으로 소거하는 기능, 선의 너비를 선택하는 기능, SetWorldTransform( )이 지원하는 다른 변환기능 그리고 [Option]차림표에 [Undo]를 추가하고 그것으로 방금 진행한 그리기처리를 취소하는 기능.

넘기기방식과 보임창

이 장을 끝내기에 앞서 Windows 2000 의 도형체계의 두가지 기능을 추가적으로 설명해 둔다. 그것은 *넘기기방식*과 *보임창*이다.

본문이나 도형관련의 API 함수에서 *론리단위*가 사용된다는것은 이미 배웠다. 론리단위는 객체가 표시될 때 Windows 에 의해 *물리단위*(이것은 화소이다.)로 변환된다. 론리단위로부터 물리단위에로의 변환비률은 현재의 넘기기방식에 의해 결정된다.

그러므로 넘기기방식은 페지공간의 단위가 장치공간에서 얼마로 되는가를 결정하는 것으로 된다.(넘기기방식자체에는 앞에서 설명한 여러가지 변환을 진행하는 기능은 없다.) 넘기기방식은 원점의 위치에도 영향을 준다.

론리단위와 물리단위의 변환에 관계되는 속성에는 넘기기방식만이 아니라 다른 두가지 속성이 더 있다. 첫 속성은 선택한 론리단위로 창문의 너비와 높이를 정의하는것이다. 두번째 속성은 몇가지 넘기기방식에서 보임창의 물리적크기를 결정하는것이다.

보임창이란 장치공간에서 정의되는 4 각형 령역이다. 보임창의 크기는 론리단위로부터 물리단위에로의 축척률을 결정하는 요인으로 된다. 출력은 보임창의 경계선내에 들어가도록 넘기기된다. 거의 모든 넘기기방식에서는 보임창의 크기가 미리 정의되여 있으므로 변경할수 없다. 그러나 보임창의 크기를 결정할수 있는 두가지 넘기기방식이 있다. 보임창의 원점을 설정할수도 있다.

넘기기방식의 설정

체계설정으로 론리단위는 화소와 같다. 그러므로 론리단위로부터 화소에로의 변환비률은 체계설정으로 1 대 1 이다. 그러나 넘기기방식을 변경하여 이 변환비률을 바꿀수 있다. 현재의 넘기기방식을 변경하려면 *SetMapMode( )*를 사용한다. 선언은 다음과 같다.

```
int SetMapMode(HDC hdc, int mode);
```

hdc 에는 장치상황의 손잡이를 설정한다. mode 에는 새로운 넘기기방식을 설정한다. 넘기기방식은 아래의 값중에서 임의의 값을 설정한다.

넘기기방식	의 미
MM_ANISOTROPIC	각 론리단위는 임의로 축척된 자리표축상의 임의의 단위로 넘기기된다.
MM_HIENGLISH	각 론리단위는 0.001inch 로 넘기기된다.
MM_HIMETRIC	각 론리단위는 0.01mm 로 넘기기된다.
MM_ISOTROPIC	각 론리단위는 균등하게 축척된 임의의 단위로 넘기기된

	다.(즉 X 축과 Y 축의 축척이 같다.)
MM_LOMETRIC	각 론리단위는 0.1mm 로 넘기기된다.
MM_LOENGLISH	각 론리단위는 0.01inch 로 넘기기된다.
MM_TEXT	각 론리단위는 장치의 1 화소에 넘기기된다.
MM_TWIPS	각 론리단위는 1/20 포인트(약 1/1440inch)로 넘기기된다.

*MM_TEXT*에서는 X축의 방향이 오른쪽 방향으로 되고 Y축의 방향이 아래방향으로 된다. *MM_ANISOTROPIC*와 *MM_ISOTROPIC*에서는 X축과 Y축의 방향을 임의로 설정할수 있다. 기타의 넘기기방식에서는 일반직각자리표계와 같이 X축의 방향이 오른쪽 방향이며 Y축의 방향이 웃방향으로 된다.

SetMapMode( )는 함수를 호출하기 전의 넘기기방식을 돌려 주며 오유가 발생한 경우에는 령을 돌려 준다. 체계설정의 넘기기방식은 MM_TEXT 이다.

현재의 넘기기방식을 변경하는데는 몇가지 목적이 있다. 례하면 프로그람의 출력을 물리단위로 표시하려는 경우에는 MM_LOMETRIC 등과 같이 현실세계와 꼭 같은 방식을 선택한다.

프로그람에서 표시하는 정보내용의 형식에 가장 부합되게 단위를 정의하려는 경우도 있다. 표시되는 정보내용의 축척을 변경하려는 경우도 있다.(출력내용의 크기를 확대하거나 축소하려는 경우이다.) X축과 Y축의 축척을 같게 하려는 경우도 있다. 이렇게 하면 X축과 Y축에서 표시되는 단위가 물리적으로 같게 된다.)

창문범위의 정의

MM_ISOTROPIC 또는 *MM_ANISOTROPIC*의 어느 한 방식을 선택한 경우에는 *창문의 범위*를 론리단위로 정의하여야 한다.(MM_ISOTROPIC 와 MM_ANISOTROPIC 에서는 임의의 단위를 사용하므로 제한을 정의하여야 한다.) 창문의 X 축방향 및 Y 축방향의 범위를 정의하자면 *SetWindowExtEx( )*함수를 사용하여야 한다. 선언은 다음과 같다.

```
BOOL SetWindowExtEx(HDC hdc, int Xextent,int Yextent,
                        LPSIZE lpOldSize);
```

hdc에는 장치상황의 손잡이를 설정한다. Xextent와 Yextent에는 새로운 X축방향과 Y 축방향의 범위를 론리단위로 설정한다. 이 함수를 호출하기 전의 창문의 범위가 lpOldSize 가 가리키는 Size 구조체에 돌려 진다. 그러나 lpOldSize 가 NULL 인 경우는 마지막으로 설정되였던 범위가 돌려 지지 않는다. 호출이 성공하면 함수는 령이 아닌 값을 돌려 주며 실패하면 령을 돌려 준다. SetWindowExtEx( )는 넘기기방식이 MM_ANISOTROPIC 또는 MM_ISOTROPIC 인 경우에만 사용된다.

창문의 론리크기를 변경한다고 하여 화면에 표시되는 창문의 물리크기까지 변경되지는 않는다는데 주의해야 한다. 선택한 론리단위로 창문의 크기를 정의할뿐이다.(보다 정

확히 말하면 창문에서 사용되는 론리단위와 장치에서 사용되는 물리단위(화소)의 관계를 정의할뿐이다.)

실례로 같은 창문에 론리적크기로 100×100을 줄수도 있고 50×75를 줄수도 있다. 이 두 설정의 차이는 화상이 표시될 때 론리단위로부터 화소에로의 변환비률뿐이다.

보임창의 정의

이미 설명한바와 같이 보임창은 *론리단위*를 물리장치로 어떻게 넘기기하는가를 결정하는 요인으로 되며 장치공간내에 있는 4각형구역이다. 보임창의 크기는 론리단위를 *화*소로 변환하기 위해 사용되는 변환비률의 물리적인 요소를 정의한다.(창문의 범위는 론리적인 요소를 정의한다.)

일반적인 넘기기방식에서는 보임창의 크기가 고정되여 있으므로 프로그람에서 변경시킬수 없다. 그러나 MM_ISOTROPIC 및 MM_ANISOTROPIC에서는 보임창의 크기를 변경시킬수 있다. 이 넘기기방식들에서는 론리단위로부터 화소에로의 변환비률을 설정할수 있기때문이다.

보임창은 *SetViewportExtEx( )*함수를 사용하여 정의된다. 선언은 다음과 같다.

```
BOOL SetViewportExtEx(HDC hdc, int Xextent, int Yextent,
                      LPSIZE lpOldSize);
```

hdc에는 장치상황의 손잡이를 설정한다. Xextent와 Yextent에는 X축방향 및 Y축방향의 보임창의 새로운 크기를 화소단위로 설정한다. 함수의 호출이 성공하면 령이 아닌 값을 돌려 주며 실패하면 령을 돌려 준다. 함수를 호출하기 전의 보임창의 크기가 lpOldSize가 가리키는 SIZE구조체에 돌려 진다. 그러나 lpOldSize를 NULL로 하면 바로 이전의 크기가 돌려 지지 않는다.

참고 *: SetViewportExtEx( )는 대응방식이 MM_ISOTROPIC 또는 MM_ANISOTROPIC인 때만 사용된다.*

보임창은 임의의 크기로 설정할수 있다. 창문전체를 포함하는 크기로 할수도 있고 창문보다 크게 할수도 있으며 혹은 창문의 일부분으로 할수도 있다.

출력은 페지공간(론리단위)으로부터 보임창(화소)으로 자동적으로 넘기기되며 그에 맞게 축척된다. 그러므로 보임창의 X축 방향 및 Y축 방향의 크기를 변경하는것은 론리단위로부터 화소에로의 변환비률을 변경하는것으로 된다.

이 결과로서 표시되여 있는 모든 내용의 크기가 변한다. 보임창의 크기를 크게 하면 보임창의 내용이 확대된다. 이와는 반대로 크기를 작게 하면 보임창의 내용이 축소된다. 이 기능은 화상의 축소나 확대에서 편리하게 리용된다.

보임창원점의 설정

체계설정으로는 보임창의 원점이 창문의 (0, 0)위치에 있다. 그러나 *SetViewportOrgEx( )*를 사용하면 원점의 위치를 변경할수 있다. 선언은 다음과 같다.

BOOL SetViewportOrgEx(HDC hdc, int X, int Y, LPPOINT lpOldOrg);

hdc에 장치상황의 손잡이를 설정한다. 보임창의 새로운 원점을 X, Y에 화소단위로 설정한다. 직전의 원점이 lpOldOrg가 가리키는 POINT구조체에 돌려 진다. 이 파라메터를 NULL로 하면 직전의 원점은 돌려 지지 않는다. 호출이 성공하면 함수는 령이 아닌 값을 돌려 주며 실패하면 령을 돌려 준다.

보임창의 원점을 변경하면 창문에 표시되는 화상의 위치가 변한다.

보임창과 넘기기방식의 실례프로그람

실례 9-3의 프로그람은 넘기기방식과 보임창의 실례이다. 이 프로그람에서 진행되는 처리는 매우 간단하다.

우선 WM_PAINT내에서 넘기기방식을 MM_ANISOTROPIC로 설정하고 창문의 범위를 200×200으로 설정하며 보임창의 크기를 변수 Xext및 Yext에 설정하고 보임창의 원점을 변수 Xorg 및 Yorg에 설정한다.

다음 어떤 화상을 그린다. 마우스의 왼쪽 단추를 누를 때마다 원점이 이동한다. 마우스의 오른쪽 단추를 누를 때마다 보임창의 크기가 커지게 된다. 원점을 이동하면 창문안에 그려진 화상이 이동한다. 보임창의 크기를 크게 하면 화상이 확대된다. 프로그람의 실행결과를 그림 9-4에 준다.

실례 9-3. Viewport 프로그람

```
// 넘기기방식과 보임창의 실례

#include <windows.h>
#include <cstring>
#include <cstdio>

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);

char szWinName[] = "MyWin"; // 창문클라스의 이름
// 보임창의 크기와 원점
int Xext=50, Yext=50;
int Xorg=0, Yorg=0;
```

```
int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;

  // 창문클라스를 정의한다.
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst;      // 실체의 손잡이
  wcl.lpszClassName = szWinName; // 창문클라스의 이름
  wcl.lpfnWndProc = WindowFunc;  // 창문함수
  wcl.style = 0;                 // 체계설정의 형식

  wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
  wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
  wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

  wcl.lpszMenuName = NULL; // 클라스차림표는 없다.
  wcl.cbClsExtra = 0;      // 보조기억기령역은 필요 없다.
  wcl.cbWndExtra = 0;

  // 창문의 배경색을 흰색으로 한다.
  wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

  // 창문클라스를 등록한다.
  if(!RegisterClassEx(&wcl)) return 0;

  /* 창문클라스가 등록되였으므로
     창문을 작성할수 있다. */
  hwnd = CreateWindow(
    szWinName, // 창문클라스의 이름
    "Mapping Modes and Viewports", // 제목
    WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
    CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
    CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
    CW_USEDEFAULT, // 너비는 Windows가 결정하게 한다.
```

```
    CW_USEDEFAULT, // 높이는 Windows가 결정하게 한다.
    NULL,          // 어미창문은 없다.
    NULL,          // 차림표는 없다.
    hThisInst,     // 실체의 손잡이
    NULL           // 추가파라메터는 없다.
  );

  // 창문을 표시한다.
  ShowWindow(hwnd, nWinMode);
  UpdateWindow(hwnd);

  // 통보문순환고리를 작성한다.
  while(GetMessage(&msg, NULL, 0, 0))
  {
    TranslateMessage(&msg); // 건반통보를 변환한다.
    DispatchMessage(&msg);  // Windows 2000에 조종을 넘긴다.
  }

  return msg.wParam;
}

/* 이 함수는 Windows 2000으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받아 들인다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  HDC hdc;
  PAINTSTRUCT paintstruct;
  char str[80];

  switch(message) {
    case WM_RBUTTONDOWN: // 보임창의 크기를 확장한다.
      Xext += 10;
      Yext += 10;
      InvalidateRect(hwnd, NULL, 1);
      break;
    case WM_LBUTTONDOWN: // 원점을 이동한다.
```

```
        Xorg += 10;
        Yorg += 10;
        InvalidateRect(hwnd, NULL, 1);
        break;
      case WM_PAINT:
        hdc = BeginPaint(hwnd, &paintstruct); // 장치상황을 얻는다.

        sprintf(str, "Viewport extents: %d %d, origin: %d %d",
                Xext, Yext, Xorg, Yorg);
        TextOut(hdc, 50, 0, str, strlen(str));

        // 넘기기방식을 설정한다.
        SetMapMode(hdc, MM_ANISOTROPIC);

        // 창문의 크기와 보임창의 크기를 설정한다.
        SetWindowExtEx(hdc, 200, 200, NULL);
        SetViewportExtEx(hdc, Xext, Yext, NULL);

        // 보임창의 원점을 설정한다.
        SetViewportOrgEx(hdc, Xorg, Yorg, NULL);

        // 몇개의 객체를 그린다.
        LineTo(hdc, 50, 50);
        Rectangle(hdc, 20, 20, 80, 80);
        Ellipse(hdc, 40, 50, 100, 120);

        EndPaint(hwnd, &paintstruct); // 장치상황을 해제한다.
        break;
      case WM_DESTROY: // 프로그람을 끝낸다.
        PostQuitMessage(0);
        break;
      default:
        /* 이 switch 문에서 지정된것 이외의 통보문은
           Windows 2000 에 처리를 맡긴다. */
        return DefWindowProc(hwnd, message, wParam, lParam);
    }

    return 0;
  }
```

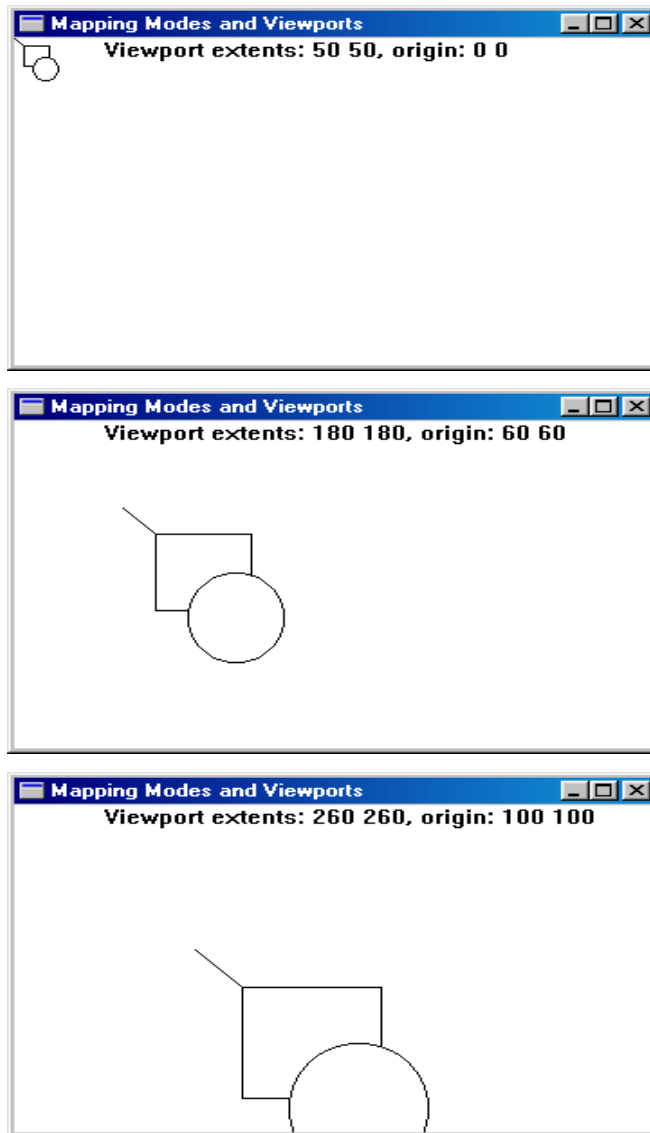



그림 9-4. 보임창프로그람의 실행결과

도형장치대면부의 실험

자리표공간, 세계변환, 보임창 및 넘기기방식 등 이 장에서 설명된 개념들가운데는 직관적으로 리해하기 힘든것들도 있다. 이 기능들과 도형장치대면부(GDI : Graphics Device Interface)전반에 숙련되기 위한 최량의 방법은 여러가지 기능을 실제로 시험해보는 간단한 프로그람들을 많이 작성해 보는것이다. 세계변환 및 기타 기능들을 잘 익혀두면 응용프로그람들에서 효률적으로 활용할수 있다.

제 10 장

공통조종체와 공통대화칸

이 장에서는 Windows 2000 의 가장 우수한 기능의 하나인 *공통조종체*에 대한 설명을 시작한다. 지금까지의 장들에서는 누름단추(Push button), 흘림띠, 검사칸 등의 표준적인 조종체들의 사용방법에 대하여 설명하였다. 이밖에도 Windows 2000 은 세련된 양식의 고급한 기능을 가진 조종체들도 제공하고 있다.

이 조종체들을 사용하면 프로그람의 사용자대면부를 보다 세련되게 할수 있다. 즉 사용자의 기대를 충분히 만족시키는 고급한 응용프로그람을 작성할수 있다. 공통조종체가 Windows 2000 프로그람작성에서 매우 중요한 요소이므로 여러장에 걸쳐 설명하기로 한다.

이 장에서는 Windows 의 두개의 공통대화칸에 대하여서도 설명한다.

*공통대화칸*이란 프로그람에서 자유로이 사용되는 미리 정의된 대화칸이며 이것을 사용하면 다음의 두가지 우점이 있다.

첫째 우점은 공통대화칸이 많은 프로그람의 입력처리에서 공통적으로 사용되는 통일적인 사용자대면부를 제공하여 준다는것이다. 두번째 우점은 공통대화칸을 사용하면 긴 프로그람코드를 서술할 필요가 없게 된다는것이다. 일반적으로 공통대화칸을 리용하는것은 같은 기능을 여러개의 공통조종체들을 리용하여 실현하는것보다 훨씬 간단하므로 프로그람작성이 보다 효률적으로 된다.

이 장에서는 처음에 공통조종체의 개념을 설명한다. 다음 가장 중요한 공통조종체의 하나인 도구띠에 대하여 설명한다. 또한 마감에 공통대화칸에 대한 설명을 한다.

공통조종체의 종류

Windows 2000 이 제공하는 공통조종체들은 다음과 같다.

조 종 체	설 명
동화조종체	AVI 파일을 상영한다.
확장복합칸조종체	확장기능을 가지는 복합칸
날자시간입력조종체	날자와 시간을 입력한다.
끌기목록칸	항목을 끌기할수 있는 목록칸
평면흘림띠	립체감이 없는 평탄한 흘림띠
머리부조종체	렬의 머리부
주목건조종체	주목건을 작성하고 지원한다.
화상목록	화상의 목록
IP 조종체	인터네트규약(IP)입력을 지원한다.
목록보기조종체	아이콘과 표식의 목록
월사업표조종체	날자입력에 사용되는 력서
페져조종체	다른 조종체를 포함한 흘리기가 가능한 조종체
진행띠	처리의 진행상태를 시각화하여 표시한다.
특성표	속성대화칸
리바조종체	다른 조종체를 포함한 띠
고급편집조종체	고급한 편집칸
상태창문	응용프로그람의 정보를 표시하는 띠
표쪽조종체	표쪽형식의 차림표(조종체의 모양이 서류철의 표쪽과 류사하다.)
도구띠	도형형식의 차림표
도구설명쪽지	튀여나오기표시되는 작은 본문칸. 일반적으로 도구띠단추나 다른 조종체의 설명을 표시하는데 사용된다.
추적띠	미끄름조절기형식의 조종체(사용방법은 흘림띠와 같으나 형태가 음량조절기와 비슷하다.)
나무구조보기조종체	나무구조로 정보를 표시한다.
오르내리기(돌리개)조종체	상하방향화살표를 제공한다. 편집칸에 결합된 경우는 돌리개조종체라고도 한다.

이 조종체들을 공통조종체라고 부르는 리유는 여러가지 응용프로그람에서 공통적으로 사용되는 미리 정의된 조종체들의 모임으로서 제공되고 있기때문이다. 모든 조종체들을 취급하는것은 불가능하므로 이 책에서는 가장 중요하고 대표적인 조종체들 몇개만을 추려서 설명하려고 한다.

이식과 관련한 요점 : 공통조종체는 Windows 95 및 NT4.0 이후 판에서 사용된다.

공통조종체를 리용하기 위한 준비와 초기화

공통조종체를 리용하자면 프로그람에 COMMCTRL.H 라는 표준머리부파일을 인용하여야 한다. 또한 프로그람에 공통조종체서고도 련결하여야 한다.

Microsoft Visual C++에서는 공통조종체서고의 이름이 COMCTL32.LIB 로 되여 있다. 이 서고는 자동적으로 련결되지 않으므로 련결프로그람의 추가선택항목을 설정할 필요가 있다. 다른 번역프로그람인 경우에는 해당한 지도서에서 확인해야 한다.

공통조종체를 사용하는 응용프로그람에서는 처음에 *InitCommonControlsEx( )*를 호출하여야 한다. InitCommonControlsEx( )는 동적련결서고에 보관된 공통조종체들을 탐색하여 그것들을 적재하고 공통조종체부분체계를 초기화한다. *InitCommonControlsEx( )*의 선언은 다음과 같다.

```
BOOL InitCommonControlsEx(LPINITCOMMONCONTROLSEX lpcc);
```

lpcc 는 적재 혹은 초기화할 조종체를 지정하기 위한 *INITCOMMONCONTROL SEX* 구조체의 지시자이다. 이 함수는 호출이 성공하면 0 이 아닌 값을 돌려 주고 실패하면 0 을 돌려 준다.

INITCOMMONCONTROLSEX 구조체의 정의는 다음과 같다.

```
typedef struct tagINITCOMMONCONTROLSEX {
    DWORD dwSize;
    DWORD dwICC;
}   INITCOMMONCONTROLSEX;
```

dwSize 에 *LPINITCOMMONCONTROLSEX* 구조체의 크기를 설정한다. dwICC 에는 적재할 조종체 혹은 조종체의 모임을 설정한다. 이것은 다음의 어느 한 값이여야 한다.

마 크 로	적재되는 조종체
ICC_ANIMATE_CLASS	동화조종체
ICC_BAR_CLASSES	상태띠, 도구띠, 도구설명쪽지 및 추적띠 조종체
ICC_COOL_CLASSES	리바조종체
ICC_DATE_CLASSES	날자시간입력조종체
ICC_HOTKEY_CLASS	주목건조종체
ICC_INTERNET_CLASSES	인터네트주소조종체
ICC_LISTVIEW_CLASSES	목록보기 및 머리부조종체
ICC_PAGESCROLLER_CLASS	페저조종체
ICC_PROGRESS_CLASS	진행띠
ICC_TAB_CLASSES	표쪽 및 도구설명쪽지조종체
ICC_TREEVIEW_CLASSES	나무구조보기 및 도구설명쪽지조종체
ICC_UPDOWN_CLASSES	오르내리기조종체
ICC_USEREX_CLASSES	확장복합칸조종체
ICC_WIN95_CLASSES	Windows 95에서 지원되는 공통조종체

실례로 아래의 프로그람코드는 도구띠조종체를 적재하고 초기화한다.

```
INITCOMMONCONTROLSEX cc;

cc.dwSize = sizeof(INITCOMMONCONTROLSEX);
cc.dwICC = ICC_BAR_CLASSES;
InitCommonControlsEx(&cc);
```

기본창문이 등록된 다음 InitCommonControlsEx( )를 호출한다.

이식과 관련한 요점 : InitCommonControlsEx( )는 종래의 InitCommonControls( )를 대신하여 쓰인다.

공통조종체는 창문이다

모든 공통조종체는 새끼창문이며 CreateWindow( ), CreateWindowsEx( ) 혹은

조종체 전용의 API 함수들중 어느 하나를 호출하여 작성된다.(CreateWindowsEx( )인 경우에는 확장형식을 설정할수도 있다.) 공통조종체는 창문이기때문에 기본적으로 지금까지 프로그람에서 일반적인 창문을 작성하던것과 같은 방법으로 취급된다.

많은 공통조종체는 사용자에 의해 조작되면 프로그람에 WM_COMMAND 또는 *WM_NOTIFY* 중의 어느 한 통보문을 보낼수 있다. *SendMessage( )*라는 API 함수를 사용하여 프로그람에서 공통조종체에 통보문을 보낼수도 있다. 선언은 다음과 같다.

```
LRESULT SendMessage(HWND hwnd, UINT Msg,
                    WPARAM wParam, LPARAM lParam);
```

hwnd 에는 공통조종체의 손잡이 , Msg 에는 조종체에 보내는 통보문, wParm 과 lParam 에는 통보문과 관련된 추가정보를 설정한다. 이 함수는 조종체로부터 응답이 있으면 그것을 돌림값으로 돌려 준다.

도구띠의 사용방법

*도구띠*는 가장 중요한 조종체의 하나이다. 그것은 도구띠가 마우스로 차림표를 선택하는 처리를 효률화해 주기때문이다. 도구띠는 형상적인 단추로 되는 아이콘을 차림표의 항목으로서 제공하므로 도형적차림표라고도 한다.

도구띠가 일반적인 차림표에 대응되는 경우가 많다. 다시말하여 차림표를 선택하는 조작의 대용으로 되는것이다. 어떤 의미에서 도구띠는 마우스를 사용한 차림표의 가속기라고도 한다.

도구띠를 작성하기 위해서는 *CreateToolbarEx( )*함수를 사용한다. 선언은 다음과 같다.

```
HWND CreateToolbarEx(HWND hwnd, DWORD dwStyle, UINT ID,
                     int NumBitmaps, HINSTANCE hInst,
                     UINT BmpID, LPCTBUTTON Buttons,
                     int NumButtons,
                     int ButtonWidth, int ButtonHeight,
                     int BmpWidth, int BmpHeight,
                     UINT Size);
```

hwnd는 도구띠를 가지는 어미창문의 손잡이이다. Style에는 도구띠의 형식을 설정한다. 도구띠의 형식에는 WS_CHILD 를 포함시켜야 한다. 또한 *WS_BORDER* 나

WS_VISIBLE 등의 표준적인 형식들을 포함시킬 수도 있다. 도구띠의 형식에는 몇가지 추가선택들이 있다. 다음에 흔히 사용되는 추가선택들을 보여 주었다.

추가선택	의　　미
TBSTYLE_TOOLTIPS	도구설명쪽지를 가진다.
TBSTYLE_WRAPABLE	긴 도구띠를 여러 행에 걸쳐 표시한다.
TBSTYLE_FLAT	평평한 도구띠로 한다.
TBSTYLE_TRANSPARENT	투명한 도구띠로 한다.

ID에는 도구띠를 식별하는 값을 설정한다. BmpID에는 도구띠에 표시되는 비트매프 자원을 식별하는 값을 설정한다.이 비트매프는 도구띠에 포함되여 있는 모든 화상들을 포함한다.

NumBitmap에 BmpID에 지정된 비트매프에 포함되는 화상의 수를 설정한다. hInst에는 응용프로그람의 실체의 손잡이를 설정한다. BmpID 에는 자원을 식별하는 값이 아니라 비트매프의 손잡이를 설정해도 된다. 이 경우에는 hInst에 NULL을 설정한다.

Buttons에 정의되는 TBBUTTON 구조체의 배렬에는 매개 단추의 정보를 설정한다. NumButtons에는 도구띠안의 단추수를 설정한다. ButtonWidth와 ButtonHeight에는 단추의 너비와 높이를 설정한다. BmpWidth와 BmpHeight에는 매개 단추안에 표시되는 화상의 너비와 높이를 설정한다. 크기를 표시하는 단위는 화소이다. Size 에는 TBBUTTON 구조체의 크기를 설정한다.

CreateToolbarEx( )는 도구띠창문의 손잡이를 돌려 준다. 호출이 실패하면 NULL을 돌려 준다.

매개 단추의 속성을 TBBUTTON 구조체에 설정한다. *TBBUTTON* 구조체의 정의는 다음과 같다.

```
typedef struct _TBBUTTON {
    int iBitmap;
    int idCommand;
    BYTE fsState;
    BYTE fsStyle;
    DWORD dwData;
    int iString;
}   TBBUTTON;
```

iBitmap 에는 단추에 표시하는 비트매프의 색인을 설정한다. 색인이 령인 단추로부터 차례로 왼쪽에서 오른쪽에로 표시된다.

idCommand 에는 단추를 식별하는 값을 설정한다. 단추가 눌리울 때마다

WM_COMMAND 통보문이 생성되여 그것이 어미창문에 전송된다. 이때 idCommand 의 값이 wParam 의 아래단어에 보관된다.

fsState 에는 단추의 초기상태를 보관한다. 이것은 다음의 값(또는 그의 조합)으로 한다.

상　　태	의　　미
TBSTATE_CHECKED	단추를 누른 상태로 한다.
TBSTATE_ELLIPSES	단추를 모두 표시할수 없는 경우에 생략기호를 표시한다.
TBSTATE_ENABLE	단추를 누를수 있다.
TBSTATE_HIDDEN	단추를 비표시로 하여 사용할수 없게 한다.
TBSTATE_INDETERMINATE	단추를 회색표시로 하여 사용할수 없게 한다.
TBSTATE_MARKED	단추에 표식을 붙인다.(눌리워 진 상태와는 다르다.)
TBSTATE_PRESSED	단추를 누른 상태로 한다.
TBSTATE_WRAP	단추를 모두 표시할수 없는 경우에 여러개의 행에 표시한다.

FsStyle 에는 단추의 형식을 설정한다. 이것은 다음에 표시한 값들의 조합으로 한다.

격　　식	의　　미
BTNS_AUTOSIZE	단추의 크기를 본문에 맞춘다.(낡은 콤파일러에서는 TBSTYLE_AUTOSIZE 를 사용한다.)
BTNS_BUTTON	표준적인 단추(낡은 콤파일러에서는 TBSTYLE_BUTTON 을 사용한다.)
BTNS_CHECK	단추가 눌리울 때마다 검사상태와 미검사상태를 반전절환한다.(낡은 콤파일러에서는 TBSTYLE_CHECK 를 사용한다.)
BTNS_CHECKGROUP	그룹안에서 하나밖에 선택할수 없는 단추로 한다.(낡은 콤파일러에서는 TBSTYLE_CHECKGROUP 을 사용한다.)
BTNS_DROPDOWN	내리펼침형식의 단추로 한다.(낡은 콤파일러에서는 TBSTYLE_GROUP 를 사용한다.)
BTNS_GROUP	하나밖에 선택되지 않는 표준적인 단추로 한다.(낡은 콤파일러에서는 TBSTYLE_GROUP 를 사용한다.)
BTNS_NOPREFIX	가속기로 되는 앞붙이를 표시하지 않는다.(낡은

	콤파일러에서는 TBSTYLE_NOPREFIX 를 사용한다.)
BTNS_SEP	분리기로 되는 단추로 한다.(이 형식의 경우는 idCommand 의 값을 령으로 한다. 낡은 콤파일러에서는 TBSTYLE_SEP 를 사용한다.)
BTNS_SHOWTEXT	단추에 본문을 표시한다.(Windows 2000 에 추가된것)
BTNS_WHOLEDROPDOWN	단추에 내리펼침을 의미하는 아래방향화살표를 붙인다.(Windows 2000 에 추가된것)

형식의 이름들을 보면 그 의미를 알수 있을것이다. 그러나 약간한 보충적설명이 필요한것들도 있다. BTNS_SEP 는 도구띠의 매개 단추사이의 간격을 비우므로 이것으로 단추들을 그룹화할수 있다.

BTNS_DROPDOWN 및 *BTNS_WHOLEDROPDOWN* 은 내리펼침차림표를 제공하는 단추를 작성하기 위해 사용된다. 이 단추들이 눌리우면 WM_COMMAND 통보문이 아니라 TBN_DROPDOWN 통보문이 생성된다. 이 통보문의 처리에는 제19장에서 설명하는 차림표의 고급한 기능이 필요된다.

이식과 관련한 요점 : 도구띠의 형식을 정의하는 BTNS 로 시작하는 마크로는 종래의 TBSTYPE 로 시작하는 마크로를 치환한것이다. 실례로 BTNS_SEP 는 TBSTYLE_SEP 로 불리워왔다. 지금은 두 종류의 마크로가 다 지원되고 있지만 BTNS 류의 마크로를 사용할것을 권고한다. 그러나 낡은 콤파일러를 사용하고 있다면 반드시 TBSTYLE 을 사용해야 하는 경우도 있다.

TBBUTTON 구조체의 dwData 에 사용자정의자료를 설정한다. iString 에는 단추와 관련된 추가선택문자렬의 색인을 설정한다. 이 통보문을 사용하지 않는 경우에는 령을 설정한다.

체계설정에서 도구띠는 완전히 자동화된 조종체이며 그것을 조종하는 프로그람코드를 서술할 필요는 없다. 그러나 몇가지 통보문을 발송하여 도구띠를 세밀하게 조종할수도 있다. 이 통보문들은 SendMessage( )를 사용하여 도구띠에 전송된다. 도구띠에 보내는 주요 통보문들은 다음과 같다.

통 보 문	의 미
TB_CHECKBUTTON	도구띠의 단추를 눌리운 상태 혹은 눌리우지 않은 상태로 한다. wParam 에 단추를 식별하는 값을 설정한다. lParam 에 령 아닌 값을 설정하면 눌

	리운 상태로 되며 령을 설정하면 눌리우지 않은 상태로 된다.
TB_ENABLEBUTTON	도구띠의 단추를 유효 혹은 무효로 한다. WParam 에는 단추를 식별하는 값을 설정한다.lParam 에 령이 아닌 값을 설정하면 유효로 되며 령을 설정하면 무효로 된다.
TB_HIDEBUTTON	도구띠의 단추를 비표시 혹은 표시된 상태로 한다.wParam 에는 단추를 식별하는 값을 설정한다. LParam 에 령이 아닌 값을 식별하면 비표시로 되며 령을 설정하면 표시된다.

도구띠는 도구띠와 관련된 정보를 프로그람에 알려 주기 위해 통지문을 생성할수 있다. 단순한 도구띠에서는 프로그람에서 통지문을 처리할 필요가 없다.(이 통지문들은 TBN_으로 시작되는 이름을 가진다. 통보문들의 의미는 머리부파일 COMMCTRL.H 혹은 API 참고서를 참조할것)

도구설명쪽지의 추가

도구띠가운데서 단추우에 1s 정도 마우스지시자를 놓아 두면 본문을 표시하는 작은 창문이 자동적으로 표시되는데 이 작은 창문을 *도구설명쪽지*라고 한다. 프로그람의 기능에는 무관계하지만 도구띠에는 도구설명쪽지가 표시되도록 하는것이 좋다.

도구띠에 도구설명쪽지를 표시하려면 도구띠를 작성할 때 형식에 *TBSTYLE_TOOLTIPS* 를 추가하여야 한다. 이렇게 하면 마우스지시자가 단추우에 약 1s 정도 놓여있을 때 도구띠가 *WM_NOTIFY* 통보문을 생성하게 된다.

일반적으로 WM_NOTIFY 통보문은 조종체가 어떤 사건의 발생을 알려 주기 위해 생성하는 통보문이다. 도구설명쪽지의 경우에는 도구설명쪽지의 본문을 표시할 필요가 있다는것을 알려 주기 위해 생성된다. 이때 lParam 에는 *NMTTDISPINFO 구조체*의 지시자가 보관되여 있다. 이 구조체의 정의는 다음과 같다.

```
Typedef struct tagNMTTDISPINFO {
      NMHDR hdr;
      LPSTR lpszText;
      char szText[80];
      HINSTANCE hinst;
      UINT uFlags;
      LPARAM lParam;
}     NMTTDISPINFO;
```

NMTTDISPINFO의 첫 성원은 *NMHDR 구조체*로 되여 있다. 정의는 다음과 같다.

```
typedef struct tagNMHDR
{
    HWND hwndFrom;          // 조종체의 손잡이
    UINT idFrom;            // 조종체의 ID
    UINT code;              // 통지코드
}   NMHDR;
```

도구설명쪽지의 표시가 요구될 때는 code에 TTN_GETDISPINFO라는 통지문이 보관되며 idFrom 에 도구설명쪽지를 표시할 단추의 ID 가 보관된다. 다른 조종체가 WM_NOTIFY 통보문을 생성하는 경우가 있으며 혹은 도구띠가 다른 종류의 통지문을 생성하는 경우도 있으므로 이 통보문들을 참조하여 어떤 사건이 발생하였는가를 식별해야 한다.

이식과 관련한 요점 : NMTTDISPINFO 구조체는 TOOLTIPTEXT 구조체로 불리우던것이다. 낡은 이름의 구조체를 사용할수도 있지만 새로운 프로그람코드에서는 NMTTDISPINFO 를 사용해야 한다. TTN_GETDISPINFO 통보문은 TTN_NEEDTEXT라고 불리우던것이다. 낡은 이름의 통보문을 사용할수도 있으나 새로운 프로그람코드에서는 TTN_GETDISPINFO를 사용해야 한다.

목적하는 도구설명쪽지를 표시하는데는 세가지 방법이 있다. 도구설명쪽지의 본문을 NMTTDISPINFO 의 szText 에 복사하는 방법, 본문의 지시자를 lpszText 에 설정하는 방법 및 문자렬자원의 자원 ID를 설정하는 방법이다.

문자렬자원을 리용하는 경우에는 문자렬의 ID 를 lpszText 에 설정하고 문자렬자원을 포함하는 실체의 손잡이를 hinst 에 설정한다.(이것은 일반적으로 프로그람실체의 손잡이이다.)

이가운데서 제일 간단한 방법은 프로그람안에 있는 문자렬의 지시자를 lpszText 에 설정하는것이다. 실례로 다음의 프로그람코드는 뒤에서 보여 주는 실례프로그람에서 도구설명쪽지의 표시요구에 응답하는 부분이다.

```
case WM_NOTIFY: // 도구설명쪽지의 표시요구에 응답한다.
      TTtext = (LPNMTTDISPINFO) lParam;
      if(TTtext->hdr.code == TTN_GETDISPINFO)
        switch(TTtext->hdr.idFrom) {
          case IDM_OPEN: TTtext->lpszText = "Open File";
```

```
                break;
              case IDM_UPPER: TTtext->lpszText = "Uppercase";
                break;
              case IDM_LOWER: TTtext->lpszText = "Lowercase";
                break;
              case IDM_SAVE: TTtext->lpszText = "Save File";
              break;
              case IDM_HELP: TTtext->lpszText = "Help";
                break;
            }
          break;
```

도구설명쪽지의 본문이 설정되고 Windows 에 조종이 넘어 가면 도구설명쪽지가 자동적으로 표시된다. 프로그람에서 이이상의 처리는 필요되지 않는다. 도구설명쪽지의 처리는 보통 자동화되여 있으며 간단히 도구띠에 표시할수 있도록 되여 있다.

NMTTDISPINFO 의 uFlags 통보문은 hdr 의 idFrom 성원에 ID 와 손잡이가운데 어느것이 보관되여 있는가를 식별하기 위한것이다. UFlags에 *TTF_IDISHWND*가 설정되여 있는 경우에는 idFrom 에 도구설명쪽지를 요구하는 조종체의 손잡이가 보관되여 있다. 그렇지 않은 경우는 idFrom 에 조종체의 ID 가 보관되여 있다. 도구띠의 도구설명쪽지에서는 idFrom 에 보통 ID 가 보관되여 있으므로 uFlags 의 값을 확인할 필요는 없다.

NMTTDISPINFO 의 lParam 통보문에는 사용자가 설정한 임의의 자료가 보관되여 있다.

도구띠의 비트매프를 작성

도구띠를 사용하기전에 화상편집기를 사용하여 매개 단추의 내부에 표시될 화상의 비트매프를 작성하여야 한다.

도구띠에 설정되는 비트매프는 한개뿐이며 하나의 비트매프안에 모든 단추의 화상을 보관하여야 한다. 실례로 도구띠의 화상이 16 × 16bit 의 크기이고 도구띠에 6 개의 단추가 있다면 도구띠에 설정하는 비트매프의 크기는 96bit(16bit × 6 개)의 너비와 16bit 의 높이로 된다.

이 장에서 작성하는 도구띠의 실례프로그람에서는 5 개의 화상이 요구된다. 매개 화상의 크기는 16 × 16bit 이다. 그러므로 80 × 16bit 의 비트매프를 작성해야 한다. 그림 10-1 은 이 장의 실례프로그람에서 사용되는 도구띠의 비트매프를 화상편집기로 작성한 것이다. 이 비트매프를 TOOLBAR.BMP 라는 파일이름으로 보관해 둔다.

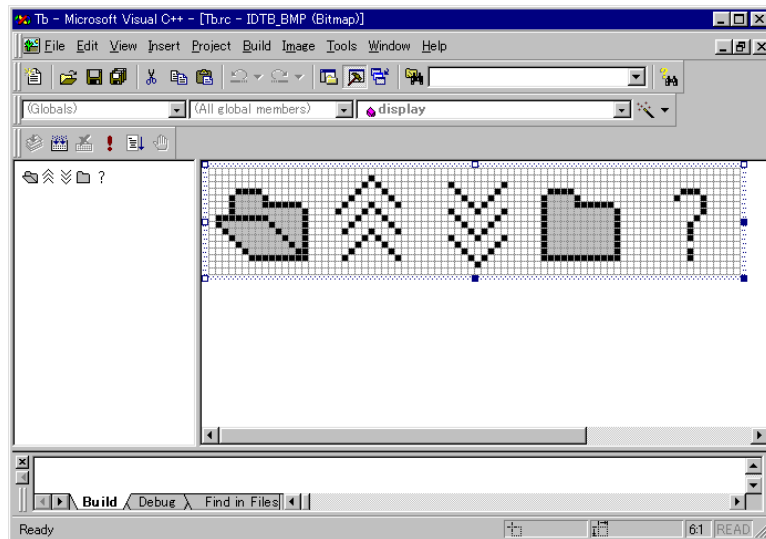



그림 10-1. 도구 비트매프의 작성

도구띠의 실례프로그람

아래의 프로그람코드는 도구띠의 응용실례를 보여 주는 프로그람이다. 이 프로그람은 본문파일용의 간단한 상용프로그람이다. 이 프로그람은 다음과 같은 기능을 제공한다.

- 파일을 읽어 내여 표시한다.
- 파일의 내용을 대문자로 변환한다.
- 파일의 내용을 소문자로 변환한다.
- 파일을 보관한다.

파일을 읽어 들이면 그 내용이 창문의 의뢰자구역에 표시된다. 파일의 내용을 흘리면서 참조할수 있도록 수직흘림띠가 표시된다. 도구띠를 차림표대신에 사용할수 있으며 도구설명쪽지도 표시된다.

파일을 읽어 들이거나 보관할 때 파일이름을 설정하기 위하여 두가지 종류의 공통대화칸을 사용한다. 이것들은 표준적인 [파일을 열기] 및 [이름을 붙여 보관] 대화칸으로서 각각 *GetOpenFileName( )* 및 *GetSaveFileName( )*라는 API 함수를 호출하여 표시된다. 이 함수들의 사용방법에 대해서는 이 장의 뒤부분에서 설명한다. 실례 10-1 에 도구띠프로그람을 표시하였다.

실례 10-1. Tb 프로그람

```
/* 도구설명쪽지기능을 가진 도구띠실례프로그람

   이 프로그람은 파일상용프로그람으로서의 몇가지
    기능을 제공한다.

*/

#include <windows.h>
#include <commctrl.h>
#include <cstring>
#include <cstdio>
#include <cctype>
#include "tb.h"

#define NUMBUTTONS 6
#define MAXSIZE 25000

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);

void InitToolbar( ); // 도구띠를 초기화한다.

void display(int startY, HDC hdc); // 파일의 내용을 표시한다.

char szWinName[] = "MyWin"; // 창문클라스의 이름

TBBUTTON tbButtons[NUMBUTTONS];

HWND tbwnd; // 도구띠의 손잡이

FILE *fp; // 파일지시자
char buf[MAXSIZE];  // 파일의 내용을 보관하는 완충기억기

TEXTMETRIC tm;
```

```
SIZE size;

int SBPos = 0; // 흘림띠의 위치
int X = 5, Y = 32; // 흘리는 크기
int NumLines = 0;
int ToolBarActive = 1;

HWND hwnd;

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  MSG msg;
  WNDCLASSEX wcl;
  HACCEL hAccel;
  INITCOMMONCONTROLSEX cc;

  // 창문클라스를 정의한다.
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst;      // 실체의 손잡이
  wcl.lpszClassName = szWinName; // 창문클라스의 이름
  wcl.lpfnWndProc = WindowFunc;  // 창문함수
  wcl.style = 0;                 // 체계설정의 형식

  wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
  wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
  wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

  wcl.lpszMenuName = "ShowFileMenu"; // 기본차림표

  wcl.cbClsExtra = 0; // 보조기억기령역은 필요 없다.
  wcl.cbWndExtra = 0;

  // 창문의 배경색을 흰색으로 한다.
  wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

  // 창문클라스를 등록한다.
```

```
    if(!RegisterClassEx(&wcl)) return 0;

    /* 창문클라스가 등록되였으므로
       창문을 작성할수 있다. */
    hwnd = CreateWindow(
      szWinName, // 창문클라스의 이름
      "Using a Toolbar", // 제목
      WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
      CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
      CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
      CW_USEDEFAULT, // 너비는 Windows가 결정하게 한다.
      CW_USEDEFAULT, // 높이는 Windows가 결정하게 한다.
      NULL,          // 어미창문은 없다.
      NULL,          // 차림표는 없다.
      hThisInst,     // 실체의 손잡이
      NULL           // 추가파라메터는 없다.
    );

    // 건반가속기를 적재한다.
    hAccel = LoadAccelerators(hThisInst, "ShowFileMenu");

    // 공통조종체를 초기화한다.
    cc.dwSize = sizeof(INITCOMMONCONTROLSEX);
    cc.dwICC = ICC_BAR_CLASSES;
    InitCommonControlsEx(&cc);

    InitToolbar(); // 도구띠를 초기화한다.

    // 도구띠를 작성한다.
    tbwnd = CreateToolbarEx(hwnd,
                          WS_VISIBLE | WS_CHILD |
                          WS_BORDER | TBSTYLE_TOOLTIPS,
                          IDM_TOOLBAR,
                          NUMBUTTONS,
                          hThisInst,
                          IDTB_BMP,
                          tbButtons,
                          NUMBUTTONS,
```

```
                    16, 16, 16, 16,
                    sizeof(TBBUTTON));

  // 창문을 표시한다.
  ShowWindow(hwnd, nWinMode);
  UpdateWindow(hwnd);

  // 통보문순환고리를 작성한다.
  while(GetMessage(&msg, NULL, 0, 0))
  {
    if(!TranslateAccelerator(hwnd, hAccel, &msg)) {
      TranslateMessage(&msg); // 건반통보를 변환한다.
      DispatchMessage(&msg); // Windows 2000에 조종을 넘긴다.
    }
  }

  return msg.wParam;
}

/* 이 함수는 Windows 2000으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받아 들인다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  HDC hdc;
  PAINTSTRUCT paintstruct;
  int i, response;
  SCROLLINFO si;
  OPENFILENAME fname;
  char filename[64]; // 파일이름
  static char fn[256]; // 완전경로명
  char filefilter[] = "C++\0*.CPP\0C\0*.C\0\0\0";

  LPNMTTDISPINFO TTtext;
  RECT rect;

  switch(message) {
```

```
    case WM_CREATE:
      // 본문치수를 얻어서 보관한다.
      hdc = GetDC(hwnd);
      GetTextMetrics(hdc, &tm);
      ReleaseDC(hwnd, hdc);
      break;
    case WM_NOTIFY: // 도구설명쪽지의 표시요구에 응답한다.
      TTtext = (LPNMTTDISPINFO) lParam;
      if(TTtext->hdr.code == TTN_GETDISPINFO)
        switch(TTtext->hdr.idFrom) {
          case IDM_OPEN: TTtext->lpszText = "Open File";
            break;
          case IDM_UPPER: TTtext->lpszText = "Uppercase";
            break;
          case IDM_LOWER: TTtext->lpszText = "Lowercase";
            break;
          case IDM_SAVE: TTtext->lpszText = "Save File";
            break;
          case IDM_HELP: TTtext->lpszText = "Help";
            break;
        }
      break;
    case WM_VSCROLL:
      switch(LOWORD(wParam)) {
        case SB_LINEDOWN:
          SBPos++;
          if(SBPos>NumLines) SBPos = NumLines;
          si.cbSize = sizeof(SCROLLINFO);
          si.fMask = SIF_POS;
          si.nPos = SBPos;
          SetScrollInfo(hwnd, SB_VERT, &si, 1);
          InvalidateRect(hwnd, NULL, 1);
          break;
        case SB_LINEUP:
          SBPos--;
          if(SBPos<0) SBPos = 0;
          si.cbSize = sizeof(SCROLLINFO);
          si.fMask = SIF_POS;
```

```
      si.nPos = SBPos;
      SetScrollInfo(hwnd, SB_VERT, &si, 1);
      InvalidateRect(hwnd, NULL, 1);
      break;
    case SB_PAGEDOWN:
      /* 현재창문의 크기에 맞추어
         한 페지분의 흘리기를 진행한다. */
      GetClientRect(hwnd, &rect);

      // 도구띠의 크기분의 보정을 진행한다.
      if(ToolBarActive) rect.bottom -= 32;

      // 흘리기할 행수를 계산한다.
      SBPos += rect.bottom /
               (tm.tmHeight+tm.tmExternalLeading);

      if(SBPos>NumLines) SBPos = NumLines;
      si.cbSize = sizeof(SCROLLINFO);
      si.fMask = SIF_POS;
      si.nPos = SBPos;
      SetScrollInfo(hwnd, SB_VERT, &si, 1);
      InvalidateRect(hwnd, NULL, 1);
      break;
    case SB_PAGEUP:
      /* 현재창문의 크기에 맞추어
         한 페지분의 흘리기를 진행한다. */
      GetClientRect(hwnd, &rect);

      // 도구띠의 크기분의 보정을 진행한다.
      if(ToolBarActive) rect.bottom -= 32;

      // 흘리기할 행수를 계산한다.
      SBPos -= rect.bottom /
               (tm.tmHeight+tm.tmExternalLeading);

      if(SBPos<0) SBPos = 0;
      si.cbSize = sizeof(SCROLLINFO);
      si.fMask = SIF_POS;
```

```
            si.nPos = SBPos;
            SetScrollInfo(hwnd, SB_VERT, &si, 1);
            InvalidateRect(hwnd, NULL, 1);
            break;
          case SB_THUMBTRACK:
            SBPos = HIWORD(wParam); // 현재위치를 얻는다.
            if(SBPos<0) SBPos = 0;
            si.cbSize = sizeof(SCROLLINFO);
            si.fMask = SIF_POS;
            si.nPos = SBPos;
            SetScrollInfo(hwnd, SB_VERT, &si, 1);
            InvalidateRect(hwnd, NULL, 1);
            break;
        }
        break;
      case WM_COMMAND:
        switch(LOWORD(wParam)) {
          case IDM_OPEN:
            // OPENFILENAME 구조체를 초기화한다.
            memset(&fname, 0, sizeof(OPENFILENAME));
            fname.lStructSize = sizeof(OPENFILENAME);
            fname.hwndOwner = hwnd;
            fname.lpstrFilter = filefilter;
            fname.nFilterIndex = 1;
            fname.lpstrFile = fn;
            fname.nMaxFile = sizeof(fn);
            fname.lpstrFileTitle = filename;
            fname.nMaxFileTitle = sizeof(filename)-1;
            fname.Flags = OFN_FILEMUSTEXIST | OFN_HIDEREADONLY;

            if(!GetOpenFileName(&fname)) // 파일이름을 얻는다.
              break;

            if((fp=fopen(fn, "r"))==NULL) {
              MessageBox(hwnd, fn, "Cannot Open File", MB_OK);
              break;
            }
```

```
      for(i=0; !feof(fp) && (i < MAXSIZE-1); i++) {
        fread(&buf[i], sizeof (char), 1, fp);
      }
      buf[i] = '\0';
      fclose(fp);

      // 행수를 구한다.
      for(NumLines=0, i=0; buf[i]; i++)
        if(buf[i] == '\n') NumLines++;

      // 흘림띠의 범위로 행수를 설정한다.
      si.cbSize = sizeof(SCROLLINFO);
      si.fMask = SIF_RANGE | SIF_POS;
      si.nMin = 0; si.nMax = NumLines;
      si.nPos = 0;
      SetScrollInfo(hwnd, SB_VERT, &si, 1);

      SBPos = 0;

      if(ToolBarActive) Y = 32;
      else Y = 0;

      InvalidateRect(hwnd, NULL, 1);
      break;
    case IDM_UPPER: // 파일의 내용을 대문자로 한다.
      for(i=0; buf[i]; i++) buf[i] = toupper(buf[i]);
      InvalidateRect(hwnd, NULL, 1);
      break;
    case IDM_LOWER: // 파일의 내용을 소문자로 한다.
      for(i=0; buf[i]; i++) buf[i] = tolower(buf[i]);
      InvalidateRect(hwnd, NULL, 1);
      break;
    case IDM_SAVE:
      // OPENFILENAME 구조체를 초기화한다.
      memset(&fname, 0, sizeof(OPENFILENAME));
      fname.lStructSize = sizeof(OPENFILENAME);
      fname.hwndOwner = hwnd;
      fname.lpstrFilter = filefilter;
```

```
        fname.nFilterIndex = 1;
        fname.lpstrFile = fn;
        fname.nMaxFile = sizeof(fn);
        fname.lpstrFileTitle = filename;
        fname.nMaxFileTitle = sizeof(filename)-1;
        fname.Flags = OFN_HIDEREADONLY;

        if(!GetSaveFileName(&fname)) // 파일이름을 얻는다.
          break;

        if((fp=fopen(fn, "w"))==NULL) {
          MessageBox(hwnd, fn, "Cannot Open File", MB_OK);
          break;
        }

        for(i=0; buf[i]; i++) {
          fwrite(&buf[i], sizeof (char), 1, fp);
        }

        fclose(fp);
        InvalidateRect(hwnd, NULL, 1);
        break;
      case IDM_SHOW: // 도구띠를 표시한다.
        ToolBarActive = 1;
        Y = 32; // 이전의 도구띠를 복귀한다.
        ShowWindow(tbwnd, SW_RESTORE);
        InvalidateRect(hwnd, NULL, 1);
        break;
      case IDM_HIDE: // 도구띠를 비표시로 한다.
        ToolBarActive = 0;
        Y = 0;
        ShowWindow(tbwnd, SW_HIDE);
        InvalidateRect(hwnd, NULL, 1);
        break;
      case IDM_HELP:
        // [?]단추를 눌리운 상태로 표시한다.
        SendMessage(tbwnd, TB_CHECKBUTTON,
                    (LPARAM) IDM_HELP, (WPARAM) 1);
```

```
        MessageBox(hwnd, "F2: Open\nF3: Uppercase\n"
                   "F4: Lowercase\nF5: Save\n"
                   "F6: Show Toolbar\n"
                   "F7: Hide Toolbar\nCtrl+X: Exit",
                   "File Utilities", MB_OK);

        // [?] 단추를 본래의 상태로 복귀한다.
        SendMessage(tbwnd, TB_CHECKBUTTON,
                    (LPARAM) IDM_HELP, (WPARAM) 0);
        break;
      case IDM_EXIT:
        response = MessageBox(hwnd, "Quit the Program?",
                              "Exit", MB_YESNO);
        if(response == IDYES) PostQuitMessage(0);
        break;
    }
    break;
  case WM_PAINT: // 다시그리기요구를 처리한다.
    hdc = BeginPaint(hwnd, &paintstruct); // 장치상황을 얻는다.
    display(SBPos, hdc);
    EndPaint(hwnd, &paintstruct); // 장치상황을 해제한다.
    break;
  case WM_DESTROY: // 프로그람을 끝낸다.
    PostQuitMessage(0);
    break;
  default:
    /* 이 switch 문에서 지정된것 이외의 통보문은
       Windows 2000 에 처리를 맡긴다. */
    return DefWindowProc(hwnd, message, wParam, lParam);
}

return 0;
}

// 도구띠구조체를 초기화한다.
void InitToolbar( )
{
```

```
    tbButtons[0].iBitmap = 0;
    tbButtons[0].idCommand = IDM_OPEN;
    tbButtons[0].fsState = TBSTATE_ENABLED;
    tbButtons[0].fsStyle = BTNS_BUTTON;
    tbButtons[0].dwData = 0L;
    tbButtons[0].iString = 0;

    tbButtons[1].iBitmap = 1;
    tbButtons[1].idCommand = IDM_UPPER;
    tbButtons[1].fsState = TBSTATE_ENABLED;
    tbButtons[1].fsStyle = BTNS_BUTTON;
    tbButtons[1].dwData = 0L;
    tbButtons[1].iString = 0;

    tbButtons[2].iBitmap = 2;
    tbButtons[2].idCommand = IDM_LOWER;
    tbButtons[2].fsState = TBSTATE_ENABLED;
    tbButtons[2].fsStyle = BTNS_BUTTON;
    tbButtons[2].dwData = 0L;
    tbButtons[2].iString = 0;

    tbButtons[3].iBitmap = 3;
    tbButtons[3].idCommand = IDM_SAVE;
    tbButtons[3].fsState = TBSTATE_ENABLED;
    tbButtons[3].fsStyle = BTNS_BUTTON;
    tbButtons[3].dwData = 0L;
    tbButtons[3].iString = 0;

    // 단추의 분리기
    tbButtons[4].iBitmap = 0;
    tbButtons[4].idCommand = 0;
    tbButtons[4].fsState = TBSTATE_ENABLED;
    tbButtons[4].fsStyle = BTNS_SEP;
    tbButtons[4].dwData = 0L;
    tbButtons[4].iString = 0;

    tbButtons[5].iBitmap = 4;
    tbButtons[5].idCommand = IDM_HELP;
```

```
    tbButtons[5].fsState = TBSTATE_ENABLED;
    tbButtons[5].fsStyle = BTNS_BUTTON;
    tbButtons[5].dwData = 0L;
    tbButtons[5].iString = 0;
  }

  // 파일의 내용을 초기화한다.
  void display(int startline, HDC hdc)
  {
    register int i, j;
    int linelim;
    int lines;
    char line[256];
    RECT rect;
    int tempY;

    GetClientRect(hwnd, &rect); // 창문의 크기를 얻는다.

    // 표시할 행수를 계산한다.
    lines = rect.bottom /
            (tm.tmHeight+tm.tmExternalLeading);

    // 첫행을 찾아 낸다.
    for(i=0; startline && buf[i]; i++)
      if(buf[i] == '\n')  startline--;

    tempY = Y;

    // 낡은 내용을 소거한다.
    PatBlt(hdc, X, Y, rect.right, rect.bottom, PATCOPY);

    // 파일의 내용을 표시한다.
    for(linelim=lines; linelim && buf[i]; i++) {
      for(j=0; j<256 && buf[i] && buf[i]!='\n'; j++, i++)
        line[j] = buf[i];

      if(!buf[i]) break;
```

```
      TextOut(hdc, X, Y, line, j);

      // 다음 행으로 이동한다.
      Y = Y + tm.tmHeight + tm.tmExternalLeading;
      linelim--;
    }
    Y = tempY;
  }
```

이 프로그램은 다음의 자원파일을 필요로 한다.

```
#include <windows.h>
#include "tb.h"

IDTB_BMP BITMAP toolbar.bmp

ShowFileMenu MENU
{
  POPUP "&File"
  {
    MENUITEM "&Open\tF2", IDM_OPEN
    MENUITEM "&Uppercase\tF3", IDM_UPPER
    MENUITEM "&Lowercase\tF4", IDM_LOWER
    MENUITEM "&Save\tF5", IDM_SAVE
    MENUITEM "E&xit\tCtrl+X", IDM_EXIT
  }
  POPUP "&Options"
  {
    MENUITEM "&Show Toolbar\tF6", IDM_SHOW
    MENUITEM "&Hide Toolbar\tF7", IDM_HIDE
 }
  MENUITEM "&Help", IDM_HELP
}

ShowFileMenu ACCELERATORS
{
  VK_F2, IDM_OPEN, VIRTKEY
  VK_F3, IDM_UPPER, VIRTKEY
```

```
  VK_F4, IDM_LOWER, VIRTKEY
  VK_F5, IDM_SAVE, VIRTKEY
  VK_F6, IDM_SHOW, VIRTKEY
  VK_F7, IDM_HIDE, VIRTKEY
  VK_F1, IDM_HELP, VIRTKEY
  "^X", IDM_EXIT
}
```

머리부파일 TB.H의 내용은 다음과 같다.

```
#define IDM_OPEN        100
#define IDM_UPPER       101
#define IDM_LOWER       102
#define IDM_SHOW        103
#define IDM_HIDE        104
#define IDM_SAVE        105
#define IDM_HELP        106
#define IDM_EXIT        107

#define IDM_TOOLBAR     200

#define IDTB_BMP        300
```

프로그람의 실행결과는 그림 10-2와 같다.

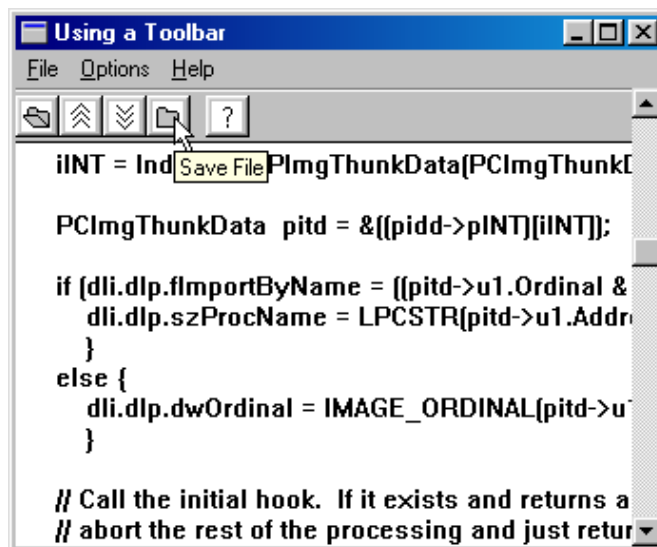



그림 10-2. 도구띠 프로그람의 실행결과

도구띠프로그람의 상세

도구띠프로그람은 매우 간단한 파일조작기능을 제공한다. 차림표 또는 도구띠로부터 [Open]이 선택된 경우는 파일이 열리고 그 내용이 buf 에 읽어 들여 진다. buf 의 크기는 적당히 설정한것이므로 필요에 따라 변경시킬수 있다.

WM_PAINT 통보문을 접수할 때마다 display( )에 의해 완충기억기(buf)의 내용이 창문에 표시된다.display( )프로그람코드는 제 8 장에서 본문을 창문에 표시할 때 리용된 것과 동일한 수법을 쓰고 있으므로 내용을 쉽게 리해할수 있을것이다.

[Uppercase]가 선택된 경우는 완충기억기의 내용이 대문자로 변환된다. [Lowercase]가 선택된 경우는 완충기억기의 내용이 소문자로 변환된다. [Save]가 선택된 경우는 완충기억기의 내용이 파일에 보관된다. 이러한 조작들은 차림표와 도구띠가운데서 임의의것을 리용하여 진행한다.

도구띠의 정보는 배렬 tbButtons 에 보관된다. 이 배렬은 InitToolBar( )안에서 초기화된다. 구조체배렬의 다섯번째 요소는 단지 단추의 분리기로 되여 있다는 점에 주의해야 한다. WinMain( )에서는 InitCommonControlsEx( )함수가 호출되고 있다. 그 다음 도구띠가 작성되여 그 손잡이가 tbwnd 에 보관된다.

도구띠의 매개 단추는 기본차림표의 차림표항목에 대응되여 있다. 분리기를 제외한 매 단추의 ID 는 대응되는 차림표의 ID 와 같다. 단추가 눌리우면 마치도 차림표가 선택된것처럼 그것의 ID 가 WM_COMMAND 통보문과 함께 전송된다. 사실 같은 case 문에서 도구띠와 차림표의 두가지 선택을 처리하고 있다.

도구띠도 창문이므로 그것을 ShowWindow( )함수를 사용하여 표시하거나 비표시로 할수 있다. 창문(여기서는 도구띠)을 비표시로 하려는 경우는 [Option]차림표에서 [Hide Toolbar]를 선택한다.다시 도구띠를 표시하려면 [Show Toolbar]를 선택한다.

도구띠는 기본창문의 의뢰자구역의 일부에 표시한것이므로 그것이 불필요한 경우에는 언제든지 소거할수 있게 해야 한다. 이것은 실례프로그람에서 볼수 있는바와 같이 아주 간단히 실현할수 있다.

IDM_HELP 의 case 문의 내용을 보면 차림표 또는 도구띠로부터 [Help]가 선택된 경우 TB_CHECKEDBUTTON 통보문을 보냄으로서 [?]단추를 눌리운 상태로 한다. 도움말통보칸이 닫기면 단추를 본래의 상태로 되돌린다.

그러므로 도움말통보칸이 표시되여 있는동안에는 [?]단추가 눌리운 상태대로 있다. 이것은 필요에 따라 프로그람에서 수동적으로 도구띠를 조종하는 하나의 실례로 된다.

창문에 표시된 파일의 내용은 수직흘림띠를 사용하여 흘리기할수 있다. 흘림띠의 사용방법에 대해서는 제 6 장에서 설명하였다. Display( )함수는 파일의 내용을 표시한다.

창문의 현재의 크기에 맞게 본문을 출력할수 있게 되여 있다.(이것은 화면에 정보를 출력하고 그것을 Windows 에 자르기시키는것보다도 효률적인 방법이다.)

창문의 현재크기는 제 8 장에서 설명한 *GetClientRect()*를 사용하여 얻는다. 이 함수는 RECT 구조체에 현재창문크기를 돌려 준다. 여기에서는 표시할 파일의 내용을 어느정도로 하면 좋겠는가를 계산하는데 이 크기를 사용하고 있다.

공통대화칸의 기초

도구띠프로그람에서는 읽거나 쓸 파일의 이름을 각각 GetOpenFileName( ) 및 GetSaveFileName( ) 라는 함수를 사용하여 얻고 있다. 이 함수들은 파일이름을 얻기 위한 [파일을 열기] 및 [이름을 붙여 보관]이라고 부르는 대화칸을 표시하는 함수들이다.

[파일을 열기] 및 [이름을 붙여 보관]은 공통대화칸이라고 부르는 미리 갖추어진 대화칸의 일종이다. *공통대화칸*은 체계에서 정의된 대화칸이며 파일이름을 얻거나 서체의 선택이나 색설정 등 다양한 프로그람에서 일반적형식의 입력을 진행하기 위하여 사용할수 있다.

다시 한보 전진

WM_SIZE 통보문을 보내기

도구띠프로그람으로 실험을 해 보자. 프로그람을 실행하고 창문의 수평방향의 크기를 늘여 보면 도구띠의 크기가 자동적으로 변하지 않는다는것을 알게 될것이다. 그 리유는 도구띠가 프로그람의 기본창문의 새끼창문이기때문이다. 그러므로 도구띠가 크기변경통보문을 자동적으로 얻게 되는것이 아니다.

만일 새끼창문(여기서는 도구띠)이 크기변경통보문을 받게 하고싶다면 통보문을 보내는 처리를 하여야 한다. 례를 들어 도구띠프로그람에서 크기변경을 실현하려면 WindowFunc( )에 다음의 case 문을 추가한다.

```
Case WM_SIZE:
    // 크기변경통보문을 도구띠에 보낸다.
    SendMessage(tbwnd, WM_SIZE, wParam, lParam);
    Break;
```

이 프로그람코드를 추가하면 기본창문을 넓힐 때 그에 응하여 도구띠의 크기가 자동적으로 변경된다.

일반적으로 기본창문의 크기가 변경된 때 창문은 *WM_SIZE* 통보문을 받는다. 창문의 새로운 너비는 LOWORD(lParam)에 보관된다. 창문의 새로운 높이는 HIWORD(lParam)에 보관된다. WParam에는 아래의 어느 한 값이 보관되여 있다.

WParam	의 미
SIZE_RESTORED	창문이 본래의 크기로 복귀되였다.
SIZE_MAXIMIZED	창문이 최대화되였다.
SIZE_MINIMIZED	창문이 최소화되였다.
SIZE_MAXHIDE	다른 창문이 최대화되였다.
SIZE_MAXSHOW	다른 창문이 본래의 크기로 복귀되였다.

일반적인 규칙으로서 프로그람에 새끼창문이 포함되여 있는 경우는 기본창문의 창문함수로부터 새끼창문에 적절한 크기변경통보문을 보낼 필요가 있다.

공통대화칸은 창문을 작성해야 하는 공통조종체와는 달리 API 함수를 호출하기만 하면 표시된다. Windows 2000에서 지원되는 공통대화칸의 종류는 다음과 같다.

함 수	표시되는 대화칸
ChooseColor( )	[색설정]대화칸을 표시한다. 사용자는 색선택이나 전용색의 작성을 진행한다.
ChooseFont( )	[서체]대화칸을 표시한다. 사용자는 서체를 선택할수 있다.
FindText( )	본문검색을 위한 [검색]대화칸을 표시한다.
GetOpenFileName( )	[파일을 열기]대화칸을 표시한다. 사용자는 읽거나 쓰려는 파일을 선택할수 있다.
GetSaveFileName( )	[이름을 붙여 보관]대화칸을 표시한다. 사용자는 정보를 보관할 파일을 선택할수 있다.
PageSetupDlg( )	인쇄페지의 서식을 설정하기 위한 [페지설정]대화칸을 표시한다.
PrintDlg( )	파일을 인쇄하기 위한 [인쇄]대화칸을 표시한다.

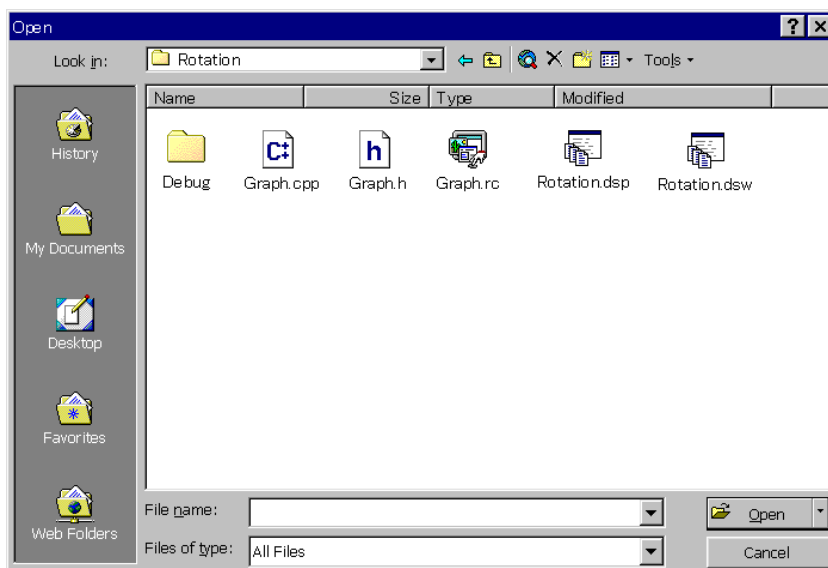
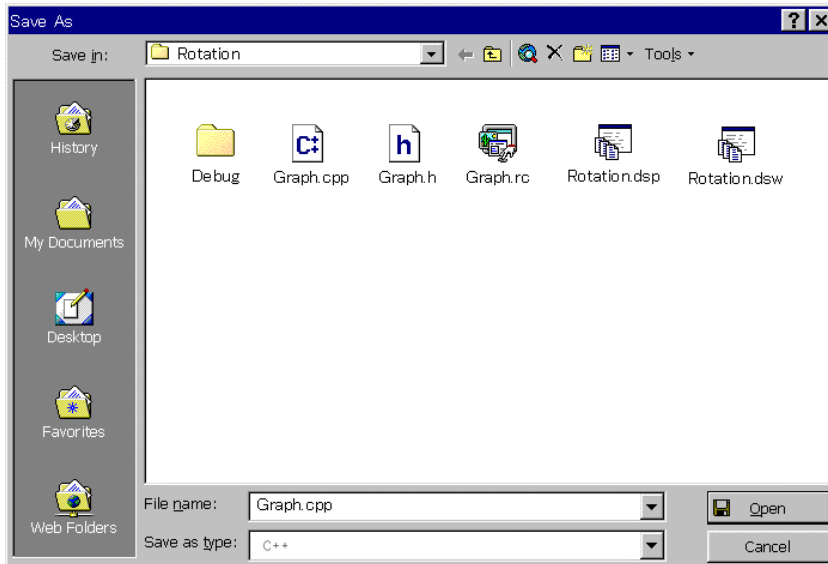
PrintDlgEx( )	Windows 2000 판의 PrintDlg( )
ReplaceText( )	본문치환을 위한 [치환]대화칸을 표시한다.

이 장의 나머지 부분에서는 [파일을 열기] 및 [이름을 붙여 보관]의 두가지 대화칸의 사용방법을 설명한다. 제 17 장에서는 [인쇄]공통대화칸의 사용방법을 설명한다. 이 공통대화칸들의 사용방법을 습득하면 자체로 다른 공통대화칸들의 사용방법도 쉽게 파악할수 있다.

공통대화칸을 통해 진행하는 형식의 입력을 자체로 작성한 대화칸에서 해서는 안된다는 리유는 없으나 그렇게 하지 않는것이 일반적이다. 공통대화칸을 리용할수 있는 상황이라면 그것을 리용해야 한다.

공통대화칸을 사용하는 리유는 그것을 프로그람의 사용자가 기대하고 있기때문이다. 공통대화칸은 꽤 복잡한 프로그람작성이 필요되는 입력처리에 매우 간단한 해결책을 제공해 준다.

GetOpenFileName( )과 GetSaveFileName( )

[Open]대화칸은 읽거나 쓸 파일의 이름을 설정하는데 사용된다. 사용자는 파일이름을 입력할수도 있고 목록에서 선택할수도 있다. 혹은 등록부를 변경하는것도 가능하다. [파일을 열기]대화칸은 GetOpenFileName( )함수의 호출에 의해서 표시된다. 이 함수가 호출되면 다음과 같은 대화칸이 표시된다.

[Save As]대화칸은 출력이 써넣어 질 파일을 설정하는데 사용된다. 다음에 [Save As]대화칸을 보여 주었다.

GetOpenFileName( ) 및 *GetSaveFileName( )*의 선언은 다음과 같다.

```
BOOL GetOpenFileName(LPOPENFILENAME lpBuf);
BOOL GetSaveFileName(LPOPENFILENAME lpBuf);
```

LpBuf 는 OPENFILENAME 구조체의 지시자이다. 이 함수들은 사용자에 의해 유효한 파일이름이 지정된 경우에 령이 아닌 값을 돌려 주며 그밖의 경우에는 령을 돌려 준다.

LpBuf 가 가리키는 OPENFILENAME 구조체는 함수를 호출하기전에 초기화하여야 한다. 돌림값으로서 사용자에 의해 지정된 파일이름과 그밖에 몇가지 정보가 OPENFILENAME 구조체에 보관된다. 아래에 *OPENFILENAME 구조체*의 정의를 준다.

```
typedef struct tagOFN
{
    DWORD lStructSize;
    HWND hwndOwner;
    HINSTANCE hInstance;
    LPCSTR lpstrFilter;
    LPSTR lpstrCustomFilter;
```

```
    DWORD nMaxCustFilter;
    DWORD nFilterIndex;
    LPSTR lpstrFile;
    DWORD nMaxFile;
    LPSTR lpstrFileTitle;
    DWORD nMaxFileTitle;
    LPCSTR lpstrInitialDir;
    LPCSTR lpstrTitle;
    DWORD Flags;
    WORD nFileOffset;
    WORD nFileExtension;
    LPCSTR lpstrDefExt;
    LPARAM lCustData;
    LPOFNHOOKPROC lpfnHook;
    LPCSTR lpTemplateName;
    void *pvReserved;// Windows 2000 에서만 사용
    DWORD dwReserved; // Windows 2000 에서만 사용
    DWORD FlagsEx;// Windows 2000 에서만 사용
} OPENFILENAME;
```

OPENFILENAME 구조체의 매 성원들을 설명해 보자.

LStructSize 에 OPENFILENAME 구조체의 크기를 설정한다. HwndOwner 에 대화칸의 어미창문의 손잡이를 설정한다. Flags 에 OFN_ENABLETEMPLATE 혹은 OFN_ENABLETEMPLATEHANDLE 을 설정한 경우는 hInstance 에 대화칸본보기의 손잡이를 설정한다. 그렇지 않은 경우 hInstance 는 사용되지 않는다.

LpstrFilter 에 파일이름의 선별기를 정의하는 문자렬쌍을 설정한다. 문자렬쌍이란 [설명]과 [마스크]이다. 실례로 "C Files""*.C"이라면 C Files 가 설명이고 마스크는 *.C 이다. 목록의 끝에는 종단을 의미하는 령문자를 두개 배치한다. 사용자가 선택할수 있는 선별기의 이름은 내리펼침형식의 목록에 표시된다.

LpstrFilter 에 NULL 을 설정한 경우는 파일이름의 선별기가 사용되지 않는다.

lpstrCustomFilter 에는 사용자에 의해 입력된 선별기를 보관하는 배렬의 지시자를 설정한다. 이 배렬에는 앞에서 보여 준 서식을 사용하여 설명과 파일선별기가 보관되게 된다. 그러나 사용자가 파일이름을 선택한후에는 새로운 선별기가 배렬에 복사된다.

lpstrCustomFilter 에 NULL 을 설정한 경우는 이 성원자체가 무시된다. 배렬의 길이는 40 문자(혹은 그이상)이여야 한다.

nMaxCustFilter 에는 lpstrCustomFilter 에서 가리키는 배렬의 크기를 설정한다. 이 값은 lpstrCustomFilter 가 NULL 인 경우에만 필요된다.

nFilterIndex 에 lpstrFilter 에서 가리키는 문자렬쌍가운데서 대화칸이 표시되였을때 초기의 파일선별기 혹은 설명으로 되는 문자렬을 설정한다. 1 이라면 맨 앞의 쌍을 가리키는것으로 되며 두번째 쌍이라면 2,… 등으로 된다. 이 값은 lpstrFilter 가 NULL 인 경우에는 무시된다. nFilterIndex 가 령인 경우에는 lpstrCustomFilter 에서 가리키는 문자렬이 사용된다.

lpstrFile 에는 사용자에 의해 선택된 완전한 파일이름,경로이름 및 구동기이름을 보관하기 위한 문자렬을 설정한다. 이 배렬에는 파일이름의 편집칸을 초기화하는데 사용되는 초기의 파일이름을 설정하든가 그렇지 않으면 빈 문자렬을 설정해 둔다.

nMaxFile 에 lpstrFile 에서 가리키는 배렬의 크기를 설정한다. 이 배렬에는 최대길이의 파일이름을 보관할수 있도록 적어도 256byte 이상의 길이로 해야 한다.

lpstrFileTitle 에는 사용자에 의해 선택된 파일의 파일이름(경로나 구동기의 정보를 제외한것)을 보관하기 위한 배렬을 설정한다. 이 파일이름이 불필요한 경우에는 이 성원을 NULL 로 한다.

nMaxFileTitle 에는 lpstrFileTitle 에서 가리키는 배렬의 크기를 설정한다.

lpstrInitialDir 에는 대화칸이 표시될 때 제일 처음 사용되는 등록부를 가리키는 문자렬을 설정한다. 그러나 lpstrFile 에 경로이름이 부여된 경우는 그 경로이름이 사용되며 lpstrInitialDir 는 무시된다. lpstrInitilalDir 가 NULL 인 경우는 현재등록부가 사용된다.

지정된 선별기에 일치되는 파일이 현재등록부에 존재하지 않는 경우는 직전의 등록부가 사용된다.(직전의 등록부는 사용자가 파일을 선택할 때마다 LastVisited 라는 체계자료기지기입항목에 보관된다.) 직전의 등록부가 없는 경우는 가능하다면 사용자의 Personal 등록부가 사용되며 그렇지 않은 경우는 탁상면등록부가 표시된다.

lpstrTitle 에는 대화칸의 제목으로 사용되는 문자렬의 지시자를 설정한다. LpstrTitle 이 NULL 인 경우는 체계설정의 제목이 사용된다. GetOpenFileName( )의 체계설정의 제목은 [Open File]이며 GetSaveFileName( )의 체계설정제목은 [Save As]이다.

Flag 성원은 대화칸에 여러가지 설정들을 진행하는데 사용된다. GetOpenFileName( )과 GetSaveFileName( )은 모두 많은 추가선택항목들을 지원하고 있다. 흔히 사용되는 추가선택항목들을 아래에 보여 주었다.(필요에 따라 두개이상의 기발을 OR 연산으로 조합할수도 있다.)

기 발	효 과
OFN_ENABLEHOOK	lpfnHook 에서 가리키는 함수를 사용하도록 허가한다.
OFN_ENABLETEMPLATE	대화칸본보기를 사용하는것을 허가한다. 이 경우는 hInstance 에 lpTemplate 에서 가리키는 대화칸을 포함한 모듈의 실체손잡이를 설정한

	다.
OFN_ ENABLETEMPLATEHANDLE	대화칸본보기를 사용하는것을 허가한다. 이 경우는 hInstance 에 대화칸을 포함하는 기억기령역의 손잡이를 설정한다.
OFN_FILEMUSTEXIST	사용자는 기존의 파일만을 선택할수 있다.
OFN_HIDEREADONLY	읽기쓰기전용의 검사칸을 표시하지 않는다.
OFN_NOCHANGEDIR	사용자는 현재의 등록부를 변경할수 없다.
OFN_OVERWRITEPROMPT	사용자가 기존의 파일을 선택한 경우에 검사를 위한 창문을 표시한다.
OFN_PATHMUSTEXIST	사용자는 기존의 경로만을 선택할수 있다.

nFileOffset 성원에는 lpstrFile 에서 가리키는 배렬에 돌려 지는 문자렬안에 들어 있는 파일이름의 제일 첫 색인을 설정한다.(이 배렬은 파일이름만이 아니라 구동기와 경로정보도 포함하고 있다.)

nFileExtension 에는 lpstrFile 에서 가리키는 배렬에 돌려 지는 문자렬안에 있는 파일의 확장자의 색인을 설정한다.

lpstrDefExt 에는 사용자에 의해 입력된 파일이름에 확장자가 포함되여 있지 않을때 사용되는 체계설정의 확장자를 설정한다.(이 확장자는 선두에 점을 붙이지 않고 설정한다.)

lCustData 에는 lpfnHook 에서 가리키는 함수에 보내는 자료를 설정한다.

lpfnHook 에는 대화칸에 보내는 통보문을 먼저 받는 함수의 지시자를 설정한다. lCustData 는 Flags 의 값에 OFN_ENABLEHOOK 가 포함되여 있는 경우에만 사용된다.

lpTemplateName 에는 대화칸본보기의 이름을 설정한다. hInstance 에는 대화칸의 자원을 포함한 모듈의 손잡이를 설정하여야 한다.

lpTemplateName 은 Flags 에 OFN_ENABLETEMPLATE 가 포함되여 있지 않은 경우에는 무시된다.

마지막 세개의 성원들인 pvReserved,dwReserved 및 FlagsEx 는 Windows 2000 에 새롭게 추가된것들로서 다른 판본의 Windows 에서는 사용되지 않는다. 현재 ExtFlags 에서 지원되는 추가기발은 OFN_EX_NOPLACESBAR 뿐이다. 이것은 탐색기(Explorer)형식의 위치띠(왼쪽에 표시되는 띠)를 비표시로 하기 위한것이다.

도구띠프로그람에서 OPENFILENAME 구조체(fname)가 어떻게 초기화되고 있는가에 주목해야 한다. 사용되지 않는 많은 성원들을 NULL 로 설정해야 하므로 맨 처음

memset( )를 사용하여 구조체의 모든 성원들을 령으로 설정하고 있다. 이것은 일반적인 기법으로서 이 프로그람에서도 사용되고 있다.

fname 의 모든 성원들을 령으로 설정한 다음 필요한 성원만을 설정한다. 배렬 filefilter 를 초기화하는 방법에 주목해야 한다. lpstrFilter 에서 가리키는 배렬에 문자렬 쌍을 설정하는 방법이 리해될것이다.

파일이름을 얻은 다음 그것을 사용하여 파일을 연다. 이 경우에는 lpstrFile 에서 가리키는 fn 에 보관되여 있는 구동기이름, 경로이름 및 파일이름전체가 사용된다. 파일을 현재등록부만으로 제한하려고 한다면 lpstrFileTitle 에서 가리키는 filename 의 내용만을 사용할수도 있다.

여러가지 추가선택항목들을 지정하여 자체로 GetOpenFileName( ) 및 GetSaveFileName( )의 기능을 구체적으로 조사해 보는것이 좋다. 례하면 lpstrCustomFilter 를 사용해 볼수 있다. 이 두개의 미리 갖추어 진 대화칸은 거의 모든 응용프로그람에서 쓰이는 중요한 대화칸들이다.

Windows 2000 프로그람에서 파일을 읽거나 쓰는 방법

이 장을 끝내기에 앞서 Windows 2000 프로그람에서 파일을 읽거나 쓰는 방법에 대해 간단히 설명해 둘 필요가 있다.

Windows 2000 프로그람에서 파일을 읽거나 쓰는것은 매우 간단하다. 왜냐하면 Windows 2000 프로그람을 작성할수 있는 모든 C/C++번역프로그람들이 Windows 2000 에서 사용할수 있는 수많은 함수와 클라스들을 제공하고 있기때문이다.

이러한것들가운데는 open( ), close( ), fopen( ), istream 및 ostream 등이 있다. 아무런 우려도 없이 이 함수나 클라스들을 여느 때처럼 사용하여 파일의 입출력을 실현할수 있다. 이러한 함수들은 도구띠프로그람에서도 사용되고 있다.

그러나 만일 필요하다면 CreateFile( ), ReadFile( ) 및 WriteFile( ) 등의 Win32 에서 정의된 API 함수들을 사용할수도 있다.

제 11 장

오르내리기조종체, 추적띠 및 진행띠

이 장에서는 Windows 2000 의 *공통조종체*들인 오르내리기조종체(Up-down control), *추적띠*(Track bar) 및 *진행띠*(Progress bar)에 대해 소개한다. 이 조종체들은 현존프로그람을 Windows 2000 에 이식할 때 매우 중요한것으로 된다. 왜냐하면 이 조종체들이 새로운 응용프로그람들에서 많이 사용되고 있으며 시각적인 효과가 크고 프로그람의 조작성을 확장할수 있기때문이다.

이 장은 이러한 조종체들에 대한 초보적지식을 설명하는것으로부터 시작한다. 다음 제 6 장에서 작성한 내려세기프로그람의 확장판의 작성을 통하여 조종체들의 사용방법을 보여 주었다. 단순히 몇개의 조종체를 추가한것에 의해 응용프로그람의 사용자대면부가 많이 개선되는것을 보게 될것이다.

오르내리기조종체의 사용방법

오르내리기조종체는 흘림띠를 간단히 한것이다. 이 조종체에는 흘림띠의 량쪽 끝의 화살표들만이 있으며 이 화살표들사이에는 띠가 없다. 어떤 Windows 응용프로그람을 사용할 때 본적이 있을수도 있겠지만 흘림띠들가운데는 너무 작아서 띠의 조작이 불가능한것도 있다. 이러한 경우에 대처하기 위해 오르내리기조종체가 고안되였다.

오르내리기조종체에는 두가지 사용방법이 있다. 하나는 독립적인 흘림띠와 류사한 사용방법이다. 다른 하나의 사용법은 련동조종체로 되는 다른 조종체와 함께 사용하는것이다. 일반적으로 편집칸을 *련동조종체*로 한다.

편집칸과 오르내리기조종체와의 조합을 *돌리개조종체*(Spin control)또는 *돌리개*(Spinner)라고 한다.오르내리기조종체의 두가지 사용방법을 아래에 보여 주었다.(왼쪽에 있는것이 돌리개조종체이다.)

돌리개조종체를 사용할 때는 조종체에 대한 대부분의 조종을 Windows 2000이 자동화하여 준다. 그러므로 응용프로그람에 간단히 돌리개조종체를 추가할수 있다. 단독의 오르내리기조종체와 돌리개조종체의 작성과 조종방법에 대해서는 다음에 설명한다.

오르내리기조종체의 작성

오르내리기조종체를 작성하려면 *CreateUpDownControl( )*함수를 사용해야 한다. 선언은 다음과 같다.

```
HWND CreateUpDownControl(DWORD Style,int X,int Y,
    int Width,int Height,HWND hParent,
    int ID,HINSTANCE hInst,
    HWND hBuddy,int Max,int Min,int StartPos);
```

Style 에 오르내리기조종체의 형식을 설정한다. 이 파라메터에는 표준적인 형식인 *WS_CHILD*, WS_VISIBLE 및 WS_BORDER 를 포함시켜야 한다.

이 형식들외에 표 11-1 에 보여 준 형식을 한개이상 설정할수 있다.

표 11-1. 오르내리기조종체의 형식

형　　식	의　　미
UDS_ALIGNLEFT	오르내리기조종체를 련동조종체의 왼쪽에 표시한다.
UDS_ALIGNRIGHT	오르내리기조종체를 련동조종체의 오른쪽에 표시한다.
UDS_ARROWKEYS	방향건을 유효로 한다.(방향건으로 조종체를 조작할수 있다.)
TBS_AUTOBUDDY	다음의 Z 차례에 있는 조종체를 련동조종체로 한다.
UDS_HORZ	오르내리기조종체를 수평으로 표시한다.(체계설정으로는 수직으로 표시된다.)
UDS_HOTTRACK	마우스가 우에 놓였을 때 오르내리기조종체의 화살표를 강조표시시킨다.(Windows 2000 및 Windows 98 에서만)
UDS_NOTHOUSANDS	큰 값이라고 해도 련동조종체에 반점을 표시하지 않는다.(돌리개조종체의 경우에만)
UDS_SETBUDDYINT	조종체의 값이 변화되였을 때 자동적으로 련동조종체의 표시를 갱신한다. 이에 의해 오르내리기조종체의 현재의 값이 련동조종체에 표시된다.
UDS_WRAP	오르내리기조종체에 표시되는 값이 긴 경우는 잘리여 표시된다.

오르내리기조종체의 위치를 X 와 Y 에 설정한다. 조종체의 너비와 높이를 Width 와 Height 에 설정한다.

hParent 에는 어미창문의 손잡이를 설정한다. ID 에는 오르내리기조종체를 식별하는 값을 설정한다. hInst 에 응용프로그람의 실체손잡이를 설정한다. 련동조종체의 손잡이는 hBuddy 에 설정한다. 련동조종체를 사용하지 않는 경우는 이 파라메터에 NULL 을 설정하여야 한다.

조종체에서 설정하는 값의 범위를 Max 와 Min 에 설정한다. Max 가 Min 보다 작은 경우는 조종체의 동작이 거꾸로 된다. 조종체의 초기값(지정된 범위의 값이여야 한다.)을 StartPos 에 설정한다.

오르내리기조종체는 화살부분이 눌리웠을 때 증가 또는 감소되는 내부적인 계수기를 가지고 있다. 내부적인 계수기의 값은 항상 조종체가 작성되였을 때 지정된 범위내에 있다.

이 함수는 조종체의 손잡이를 돌려 준다. 호출이 실패하면 NULL 이 돌려 진다.

오르내리기조종체로부터 통보문을 받아들이기

오르내리기조종체의 화살부분이 눌리우면 조종체가 수직형(체계설정형식)인가 수평형인가에 따라 조종체로부터 어미창문에 WM_VSCROLL 또는 *WM_HSCROLL* 통보문이 발송된다. 이때의 lParam에는 오르내리기조종체의 손잡이가 보관되여 있다.

WM_VSCROLL 통보문과 WM_HSCROLL 통보문을 생성하는 조종체가 한개만이 아닌 경우도 있으므로 이 손잡이의 값을 검사하여 그것이 오르내리기조종체인가를 검사할 필요가 있다.

오르내리기조종체의 새로운 값은 HIWORD(wParam)에 보관되여 있다.

오르내리기조종체에 통보문을 보내기

오르내리기조종체는 여러가지 통보문에 응답한다.기본적인 통보문들을 표 11-2에 주었다. 실례로 조종체의 값을 얻는 경우에는 *UDM_GETPOS* 통보문을 보낸다. 이 통보문에 응답하여 조종체의 현재값이 돌려 진다. 오르내리기조종체의 값을 설정하는 경우에는 *UDM_SETPOS* 통보문을 보낸다. 오르내리기조종체에 통보문을 보내려면 *SendMessage( )*함수를 사용한다.

표 11-2. 오르내리기조종체의 주되는 통보문

통 보 문	의 미
UDM_GETBUDDY	련동조종체의 손잡이를 얻는다. 돌림값으로서 손잡이가 돌려 진다. wParam과 lParam에 령을 설정한다.
UDM_GETPOS	현재의 값을 얻는다. 돌림값의 아래단어에 현재의 값이 보관된다. wParam 과 lParam 에 령을 설정한다.
UDM_GETRANGE	현재값의 범위를 얻는다. 돌림값의 아래단어에 최대값이 보관되고 아래단어에 최소값이 보관된다. wParam과 lParam에 령을 설정한다.
UDM_SETBUDDY	새로운 련동조종체를 설정한다. 직전의 련동조종체의 손잡이가 돌림값으로서 돌려 진다. wParam에 새로운 련동조종체의 손잡이를 설정하고 lParam 에 령을 설정한다.
UDM_SETPOS	현재의 값을 설정한다. wParam에 령을 설정한다. lParam에는 새로운 값을 설정한다.
UDM_SETRANGE	현재범위를 설정한다. wParam 에 령을 설정한다. lParam 의 아래단어에 새로운 최대값을 설정하며 아래단어에 새로운 최소값을 설정한다.

돌리개조종체의 작성

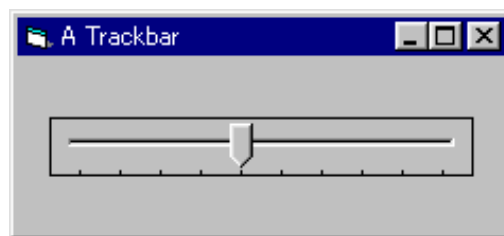
오르내리기조종체를 단독으로 사용해도 전혀 문제가 생기지 않지만 대부분의 경우 편집칸과 조합하여 사용한다. 이미 설명한것처럼 이 조합을 *돌리개조종체*(Spin control)라고 부른다. 돌리개조종체가 오르내리기조종체의 일반적인 사용방법이므로 Windows 2000 은 특별한 지원을 제공하고 있다. 즉 돌리개조종체는 완전히 자동화된 조종체이므로 프로그람에서 직접 조종할 필요는 없다.

돌리개조종체를 작성하려면 오르내리기조종체의 *련동조종체*로서 편집칸을 지정하여야 한다. 이렇게 하면 오르내리기조종체가 조작될 때마다 조종체의 새로운 값이 편집칸에 자동적으로 표시된다.

또한 편집칸의 값을 변경하면 오르내리기조종체의 내부적인 계수기도 자동적으로 변한다. 일반적으로 편집칸은 프로그람의 자원파일에서 정의된다. 이 장의 뒤부분에서 돌리개조종체의 실례를 보게 된다.

추적띠의 사용방법

*추적띠*를 *미끄름조종체*(Slider control)라고 부르는 경우도 있다. 추적띠는 시각적효과가 큰 공통조종체의 하나로서 립체단일선택 등의 전자제품에 있는 미끄름조절기에 류사한 외형을 가지고 있다. 추적띠는 조절범위내에서 이동할수 있는 미끄름조절기를 가지고 있다.

외형상차이는 있지만 프로그람에서 추적띠와 흘림띠를 조종하는 방법은 대부분 같다. 추적띠는 프로그람에서 실제의 장치를 모방할 때 특히 편리하다. 실례로 프로그람에서 록음기의 주파수평형기를 모방한다고 하면 주파수의 표시와 설정을 진행하는데 추적띠를 사용할수 있다. 다음에 추적띠의 외형을 보여 주었다.

추적띠를 작성하려면 *CreateWindow( )* 또는 *CreateWindowEx( )*를 사용한다. *CreateWindowEx( )*에서는 확장형식을 설정할수 있다. 이 장에서 작성하는 추적띠의 실례프로그람에서는 확장형식을 사용하지 않는다. 추적띠의 창문클라스는 *TRACKBAR_CLASS*이다.

추적띠의 형식

*추적띠*를 작성할 때 여러가지 형식을 설정할수 있다. 주요 형식들을 표 11-3에 주었다. 작은 선들이 표시되도록 *TBS_AUTOTICKS* 형식을 설정하는것이 일반적이다. 이 검

사표식은 띠의 눈금으로 된다.

표 11-3. 추적띠의 형식

형　　식	효　　과
TBS_AUTOTICKS	추적띠에 자동적으로 눈금을 표시한다.
TBS_HORZ	수평방향의 추적띠로 한다.(체계설정)
TBS_VERT	수직방향의 추적띠로 한다.
TBS_BOTTOM	띠의 밑에 눈금을 표시한다.(수평방향추적띠의 체계설정)
TBS_TOP	띠의 우에 눈금을 표시한다.
TBS_LEFT	띠의 왼쪽에 눈금을 표시한다.
TBS_RIGHT	띠의 오른쪽에 눈금을 표시한다.(수직방향추적띠의 체계설정)
TBS_BOTH	띠의 량쪽에 눈금을 표시한다.
TBS_TOOLTIPS	추적띠에 도구설명쪽지를 표시한다. 체계설정으로는 이 도구설명쪽지에 미끄름조절기의 현재위치가 표시된다.

추적띠에 통보문을 보내기

지금까지 설명한 다른 공통조종체들과 같이 SendMessage( )함수를 사용하여 추적띠에 통보문을 보낼수 있다. 추적띠의 주요 통보문들을 표 11-4에 보여 주었다.

표 11-4. 추적띠의 기본통보문

통 보 문	의　　미
TBM_GETPOS	현재위치를 얻는다. wParam과 lParam에 령을 설정한다.
TBM_GETRANGEMAX	추적띠의 범위의 최대값을 얻는다.wParam과 lParam에 령을 설정한다.
TBM_GETRANGEMIN	추적띠의 범위의 최소값을 얻는다.wParam과 lParam에 령을 설정한다.
TBM_SETBUDDY	추적띠의 련동조종체를 지정한다. wParam에는 련동조종체의 위치를 설정한다. 령을 설정하면 련동조종체가 오른쪽 또는 밑에 표시된다. 령이 아닌 값을 설정하면 련동조종체가 왼쪽 또는 우에 표시된다. lParam에 련동조종체로 되는 편집칸의 손잡이를 설정한다.
TBM_SETPOS	현재위치를 설정한다. 추적띠를 다시그리기하는 경

	우는 wParam 에 령이 아닌 값을 설정하고 다시그리기 하지 않는 경우에는 령을 설정한다. lParam 에는 새로운 위치를 설정한다.
TBM_SETRANGE	추적띠의 범위를 설정한다. 추적띠를 다시그리기하는 경우에는 wParam 에 령이 아닌 값을 설정하고 다시그리기하지 않는 경우에는 령을 설정한다. lParam 에는 범위를 설정한다. 최소값을 lParam 의 아래단어에 설정하고 최대값을 lParam 의 웃단어에 설정한다.
TBM_SETRANGEMAX	최대값을 설정한다. 추적띠를 다시그리기하는 경우에는 wParam 에 령이 아닌 값을 설정하고 다시그리기하지 않는 경우에는 령을 설정한다. lParam 에는 최대값을 설정한다.
TBM_SETRANGEMIN	최소값을 설정한다. 추적띠를 다시그리기하는 경우에는 wParam 에 령이 아닌 값을 설정하고 그렇지 않은 경우에는 령을 설정한다. lParam 에는 최소값을 설정한다.
TBM_SETTIPSIDE	추적띠에 도구설명쪽지를 표시할 위치를 설정한다. wParam 에는 TBTS_TOP, TBTS_BOTTOM, TBTS_LEFT 및 TBTS_RIGHT 의 어느 하나를 설정한다. lParam 에는 령을 설정한다.

추적띠에 반드시 보내야 할 통보문에는 *TBM_SETRANGE* 와 *TBM_SETPOS* 의 두가지가 있다. 이 통보문들은 각각 추적띠의 범위와 초기위치를 설정하기 위한것들이다. 이 값들을 추적띠의 작성시에는 설정할수 없다.

TBM_SETBUDDY 통보문에 주목해야 한다. 이 통보문을 사용하여 추적띠에 *련동조종체*를 결합시킬수 있다. 오르내리기조종체의 경우와 같이 추적띠의 련동조종체도 일반적으로 편집칸으로 된다.

lParam 에는 련동조종체의 손잡이를 설정한다. wParam 에는 표시위치를 설정한다. wParam 에 령을 설정하면 련동조종체의 표시위치가 수평추적띠의 오른쪽 또는 수직추적띠의 밑으로 된다. wParam 에 령이 아닌 값을 설정하면 련동조종체의 표시위치가 수평추적띠의 왼쪽 또는 수직추적띠의 웃쪽으로 된다.

추적띠에는 한개이상의 련동조종체를 지정할수 있다. 두개의 련동조종체가 사용되는 경우에는 그것들이 추적띠의 량쪽에 표시된다.(상세한 내용은 이 장의 뒤부분에 있는 [**다시 한보 전진** : 추적띠에서 련동조종체를 사용하는 방법]을 참고할것)

추적띠의 통지문의 처리

추적띠를 조작하면 추적띠가 수평형인가 수직형인가에 따라 WM_HSCROLL 과 WM_VSCROLL 중의 임의의 통보문이 생성된다. 실제 조작내용은 wParam 의 아래단어에 보관되여 있다. 이 값을 *통지문*이라고 부른다. lParam 에는 통보문을 생성한 추적띠의 손잡이가 보관되여 있다. 추적띠의 주요한 통지문들을 표 11-5 에 제시하였다.

표 11-5. 추적띠의 기본 통지문

통 보 문	의 미
TB_BOTTOM	[End]건이 눌리웠다. 미끄름조절기가 최소위치로 이동했다.
TB_ENDTRACK	추적띠의 조작이 완료되였다.
TB_LINEDOWN	오른쪽 또는 왼쪽 방향건이 눌리웠다.
TB_LINEUP	왼쪽 또는 웃쪽방향건이 눌리웠다.
TB_PAGEDOWN	[Page Down]건이 눌리웠든가 조절기의 전방부분에서 마우스가 찰칵되였다.
TB_PAGEUP	[Page Up]건이 눌리웠든가 조절기의 후방부분에서 마우스가 찰칵되였다.
TB_THUMBPOSITION	마우스를 리용하여 조절기가 이동되였다.
TB_THUMBTRACK	마우스를 리용하여 조절기가 끌기되였다.
TB_TOP	[Home]건이 눌리웠다. 조절기가 최대위치로 이동했다.

추적띠의 조종은 자동화되여 있다. 실례로 사용자가 추적띠를 조작하면 자동적으로 조절기의 표시위치가 변화되므로 이것을 프로그람에서 조작할 필요는 없다.

참고 *: 추적띠로부터 TB_THUMBTRACK 통보문 또는 TB_THUMBPOSITION 통보문을 받았을 때는 HIWORD(wParam)의 값에 띠의 현재위치가 보관되여 있다. 추적띠의 다른 통보문들에서는 이 값이 령으로 되여 있다.*

진행띠의 사용방법

*진행띠*는 처리과정의 진행상황을 표시하는 작은 창문이다. 처리의 진행상황에 맞추어 띠가 왼쪽에서부터 오른쪽으로 채색되여 나간다. 띠가 완전히 채색되면 처리과정이 끝났다는것을 의미한다.

진행띠는 공통조종체들가운데서 가장 간단한 조종체의 하나이다. 진행띠는 창문클라스에 *PROGRESS_CLASS* 를 지정하고 *CreateWindow( )*또는 *CreateWindowEx( )*를 호출하여 작성한다.

*SendMessage( )*를 사용하여 프로그람으로부터 진행띠에 통보문을 보낼수 있다. 진행띠는 통보문을 보내지 않는다. 진행띠의 범위를 설정하거나 진행상황을 갱신하려면 진행띠에 통보문을 보낸다. 진행띠의 주요 통보문들을 표 11-6 에 보여 주었다.

표 11-6. 진행띠의 기본통보문

통 보 문	의 미
PBM_SETPOS	진행띠의 위치를 지정한다. 직전의 위치가 돌려 진다. wParam 에 새로운 위치를 설정한다.lParam 에 령을 설정한다.
PBM_SETRANGE	진행띠의 범위를 설정한다. 직전의 범위가 돌려 지며 돌림값의 웃단어에는 최대값, 아래단어에는 최소값이 보관된다. wParam 에는 령을 설정한다. lParam 의 웃단어에는 최대값,아래단어에는 최소값을 보관하여 범위를 설정한다.
PBM_SETSTEP	증가값(걸음수)을 설정한다. 직전의 증가값이 돌려 진다. wParam 에 새 증가값을 설정한다.lParam 에는 령을 설정한다.
PBM_STEPIT	걸음수만큼 진행띠를 전진시킨다. wParam 과 lParam 에 령을 설정한다.

체계설정으로 진행띠의 범위는 0～100 으로 되여 있다. 그러나 이 범위는 0～65535 사이의 임의의 값으로 변경할수 있다. *PBM_STEPIT* 통보문을 보내여 진행띠의 진행상황을 변경할수 있다. 이 통보문을 보내면 진행띠의 현재위치가 사전에 정의된 걸음수만큼 전진한다. 체계설정의 걸음수는 10 으로 되여 있으나 임의의 값으로 변경시킬수 있다.

진행띠의 위치를 갱신하면 띠의 채색된 부분이 증가한다. 진행띠는 긴 처리과정의 진행상황을 표시하는 목적에 사용되는것이므로 처리과정이 다 끝났을 때는 띠의 모든 부분이 채색되도록 한다.

이전에는 진행띠에 형식을 설정할 필요가 없었으나 최근에 와서 두가지 형식이 추가되였다. 첫 형식은 *PBS_SMOOTH* 이다. 체계설정으로 진행띠의 표시는 단계적으로 변화되지만 PBS_SMOOTH 를 지정하면 진행띠의 표시가 련속적으로 원활하게 변화되게 된다.

둘째 형식은 *PBS_VERTICAL* 이다. 이것은 진행띠를 수직으로 표시시키기 위한것이다. 일반적으로 진행띠는 왼쪽에서 오른쪽으로 채색되여 나가지만 수직형의 진행띠에서는 아래에서 우로 올라 가면서 채색된다.

돌리개조종체, 추적띠 및 진행띠의 실례

제 6 장의 앞부분에서 작성한 내려세기시계 프로그람을 개조하여 돌리개조종체,추적띠 및 *진행띠*의 실례프로그람으로 만들어 보자.

이 프로그람에서는 시계의 시간간격의 s 수를 설정하는데 돌리개조종체를 사용한다. 시계의 시간간격이 경과하였을 때 울리는 경보음의 회수를 설정하는데 추적띠를 사용한다. [Beep At End]가 선택되여 있지 않는 경우는 시계의 시간간격이 경과하였을 때 통보칸이 표시된다. 진행띠에는 내려세기의 진행상황이 표시된다. 확장판내려세기시계프로그람의 완전한 프로그람코드를 실례 11-1 에 주었다.

실례 11-1. Timer 프로그람

```
/* 내려세기시계에 돌리개조종체, 추적띠 및
진행띠를 추가한다. */

#include <windows.h>
#include <commctrl.h>
#include <cstring>
#include <cstdio>
#include "timer.h"

#define BEEPMAX 10
#define MAXTIME 99

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK DialogFunc(HWND, UINT, WPARAM, LPARAM);

char szWinName[] = "MyWin"; // 창문클라스의 이름

HINSTANCE hInst;

HWND hwnd;

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
```

```
MSG msg;
WNDCLASSEX wcl;
HACCEL hAccel;
INITCOMMONCONTROLSEX cc;

// 창문클라스를 정의한다.
wcl.cbSize = sizeof(WNDCLASSEX);

wcl.hInstance = hThisInst;      // 실체의 손잡이
wcl.lpszClassName = szWinName; // 창문클라스의 이름
wcl.lpfnWndProc = WindowFunc;  // 창문함수
wcl.style = 0;                 // 체계설정의 형식

wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

wcl.lpszMenuName = "TimerMenu"; // 기본차림표

wcl.cbClsExtra = 0; // 보조기억기령역은 필요 없다.
wcl.cbWndExtra = 0;

// 창문의 배경색을 흰색으로 한다.
wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

// 창문클라스를 등록한다.
if(!RegisterClassEx(&wcl)) return 0;

/* 창문클라스가 등록되였으므로
   창문을 작성할수 있다. */
hwnd = CreateWindow(
  szWinName, // 창문클라스의 이름
  "Spin Controls, Trackbars, and Progess Bars", // 제목
  WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
  CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 너비는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 높이는 Windows가 결정하게 한다.
```

```
    NULL,          // 어미창문은 없다.
    NULL,          // 차림표는 없다.
    hThisInst,     // 실체의 손잡이
    NULL           // 추가파라메터는 없다.
  );

  hInst = hThisInst; // 현재실체의 손잡이를 보관한다.

  // 건반가속기를 적재한다.
  hAccel = LoadAccelerators(hThisInst, "TimerMenu");

  // 공통조종체를 초기화한다.
  cc.dwSize = sizeof(INITCOMMONCONTROLSEX);
  cc.dwICC = ICC_BAR_CLASSES | ICC_UPDOWN_CLASS |
             ICC_PROGRESS_CLASS;
  InitCommonControlsEx(&cc);

  // 창문을 표시한다.
  ShowWindow(hwnd, nWinMode);
  UpdateWindow(hwnd);

  // 통보문순환고리를 작성한다.
  while(GetMessage(&msg, NULL, 0, 0))
  {
    if(!TranslateAccelerator(hwnd, hAccel, &msg)) {
      TranslateMessage(&msg); // 건반통보를 변환한다.
      DispatchMessage(&msg); // Windows 2000에 조종을 넘긴다.
    }
  }

  return msg.wParam;
}

/* 이 함수는 Windows 2000으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받아 들인다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
```

```
{
  int response;

  switch(message) {
    case WM_COMMAND:
      switch(LOWORD(wParam)) {
        case IDM_DIALOG:
          DialogBox(hInst, "MYDB", hwnd, (DLGPROC) DialogFunc);
          break;
        case IDM_EXIT:
          response = MessageBox(hwnd, "Quit the Program?",
                               "Exit", MB_YESNO);
          if(response == IDYES) PostQuitMessage(0);
          break;
        case IDM_HELP:
          MessageBox(hwnd, "Try the Timer", "Help", MB_OK);
          break;
      }
      break;
    case WM_DESTROY: // 프로그람을 끝낸다.
      PostQuitMessage(0);
      break;
    default:
      /* 이 switch 문에서 지정된것 이외의 통보문은
         Windows 2000 에 처리를 맡긴다. */
      return DefWindowProc(hwnd, message, wParam, lParam);
  }
  return 0;
}

// 시계의 대화함수
BOOL CALLBACK DialogFunc(HWND hdwnd, UINT message,
                         WPARAM wParam, LPARAM lParam)
{
  char str[80];

  HDC hdc;
  PAINTSTRUCT paintstruct;
```

```
static int t;
int i;

static long udpos = 1;
static long trackpos = 1;
static HWND hEboxWnd;
static HWND hTrackWnd;
static HWND udWnd;
static HWND hProgWnd;
int low=1, high=BEEPMAX;


switch(message) {
  case WM_INITDIALOG:
    hEboxWnd = GetDlgItem(hdwnd, IDD_EB1);
    // 오르내리기조종체를 작성한다.
    udWnd = CreateUpDownControl(
                    WS_CHILD | WS_BORDER | WS_VISIBLE |
                    UDS_SETBUDDYINT | UDS_ALIGNRIGHT,
                    10, 10, 50, 50,
                    hdwnd,
                    IDD_UPDOWN,
                    hInst,
                    hEboxWnd,
                    MAXTIME, 1, udpos);

    // 추적띠를 작성한다.
    hTrackWnd = CreateWindow(TRACKBAR_CLASS,
                    "",
                    WS_CHILD | WS_VISIBLE | WS_TABSTOP |
                    TBS_AUTOTICKS | WS_BORDER,
                    2, 50,
                    200, 28,
                    hdwnd,
                    NULL,
                    hInst,
                    NULL
    );
```

```
    SendMessage(hTrackWnd, TBM_SETRANGE,
                1, MAKELONG(low, high));
    SendMessage(hTrackWnd, TBM_SETPOS,
                1, trackpos);

    // 진행띠를 작성한다.
    hProgWnd = CreateWindow(PROGRESS_CLASS,
                  "",
                  WS_CHILD | WS_VISIBLE | WS_BORDER,
                  2, 84,
                  240, 12,
                  hdwnd,
                  NULL,
                  hInst,
                  NULL);

    // 증가걸음을 1로 한다.
    SendMessage(hProgWnd, PBM_SETSTEP, 1, 0);

    // 「As-Is」단일선택단추를 선택상태로 한다.
    SendDlgItemMessage(hdwnd, IDD_RB3, BM_SETCHECK, 1, 0);

    // 「Beeps」편집칸에 수값을 설정한다.
    SetDlgItemInt(hdwnd, IDD_EB2, trackpos, 1);
    return 1;
  case WM_VSCROLL: // 10s의 설정으로 경보음을 울린다.
    if(udWnd==(HWND)lParam) // 돌리개조종체의 경우
      if(!(HIWORD(wParam) % 10)) MessageBeep(MB_OK);
    return 1;
  case WM_HSCROLL: // 추적띠가 능동
    if(hTrackWnd != (HWND)lParam) break; // 추적띠가 아닌 경우

    switch(LOWORD(wParam)) {
      case TB_TOP:
      case TB_BOTTOM:          // 이 실례프로그람에서는
      case TB_LINEUP:          // 모든 통보문들을
      case TB_LINEDOWN:        // 같은 방법으로
      case TB_THUMBPOSITION:   // 처리한다.
```

```
        case TB_THUMBTRACK:
        case TB_PAGEUP:
        case TB_PAGEDOWN:
          trackpos = SendMessage(hTrackWnd, TBM_GETPOS,
                          0, 0);
          SetDlgItemInt(hdwnd, IDD_EB2, trackpos, 1);
          return 1;
      }
      break;
    case WM_COMMAND:
      switch(LOWORD(wParam)) {
        case IDCANCEL:
          KillTimer(hdwnd, IDD_TIMER);
          EndDialog(hdwnd, 0);
          return 1;
        case IDD_EB2:
          /* 사용자가 [Beeps]편집칸에 수값을 입력하면
             추적띠를 갱신한다.  */
          trackpos = GetDlgItemInt(hdwnd, IDD_EB2, NULL, 1);
          SendMessage(hTrackWnd, TBM_SETPOS, 1, trackpos);
          return 1;
        case IDD_START: // 시계를 기동시킨다.
          SetTimer(hdwnd, IDD_TIMER, 1000, NULL);

          // 설정된 s수를 얻는다.
          t = udpos = SendMessage(udWnd, UDM_GETPOS, 0, 0);

          // 진행띠를 초기화한다.
          SendMessage(hProgWnd, PBM_SETRANGE, 0,
                       MAKELONG(0, udpos));
          SendMessage(hProgWnd, PBM_SETPOS, 0, 0);

          if(SendDlgItemMessage(hdwnd,
                     IDD_RB1, BM_GETCHECK, 0, 0))
             ShowWindow(hwnd, SW_MINIMIZE);
          else
          if(SendDlgItemMessage(hdwnd,
                     IDD_RB2, BM_GETCHECK, 0, 0))
             ShowWindow(hwnd, SW_MAXIMIZE);
```

```
            return 1;
        }
        break;
    case WM_TIMER: // 시계의 설정시간이 경과했다.
        if(t==0) {
          KillTimer(hdwnd, IDD_TIMER);

          // 경보음을 울리든가 통보칸을 표시한다.
          if(SendDlgItemMessage(hdwnd,
                      IDD_CB2, BM_GETCHECK, 0, 0)) {

            // 경보음을 울릴 때마다 추적띠를 이동시킨다.
            for(i=trackpos; i; i--) {
              MessageBeep(MB_OK);
              Sleep(1000);
              SendMessage(hTrackWnd, TBM_SETPOS, 1, i-1);
            }

            // 내려세기후에 추적띠를 재설정한다.
            SendMessage(hTrackWnd, TBM_SETPOS, 1, trackpos);
          }
          else MessageBox(hdwnd, "Timer Went Off", "Timer", MB_OK);

          SetDlgItemInt(hdwnd, IDD_EB1, udpos, 1);
          ShowWindow(hwnd, SW_RESTORE);
          return 1;
        }
        t--;

        // 진행띠를 전진시킨다.
        SendMessage(hProgWnd, PBM_STEPIT, 0, 0);

        // 내려세기상태를 표시하는가를 검사한다.
        if(SendDlgItemMessage(hdwnd,
                    IDD_CB1, BM_GETCHECK, 0, 0)) {
            SetDlgItemInt(hdwnd, IDD_EB1, t, 1);
        }
        return 1;
    case WM_PAINT:
        hdc = BeginPaint(hdwnd, &paintstruct);
        SetBkMode(hdc, TRANSPARENT);
```

```
        sprintf(str, "Seconds");
        TextOut(hdc, 44, 6, str, strlen(str));
        sprintf(str, "Beeps");
        TextOut(hdc, 186, 6, str, strlen(str));
        sprintf(str, "Set Number of Beeps");
        TextOut(hdc, 30, 32, str, strlen(str));
        EndPaint(hdwnd, &paintstruct);
        return 1;
    }

    return 0;
}
```

이 프로그람에서 사용하는 자원파일을 실례 11-2 에 준다. IDD_EB1 로 표시되는 편집칸은 돌리개조종체를 형성하도록 오르내리기조종체와 결합된다.

실례 11-2. Timer2 프로그람

```
// 돌리개조종체와 추적띠의 실례
#include <windows.h>
#include "timer.h"
TimerMenu MENU
{
  POPUP "&Options"
  {
    MENUITEM "&Timer\tF2", IDM_DIALOG
    MENUITEM "E&xit\tCtrl+X", IDM_EXIT
  }
  MENUITEM "&Help", IDM_HELP
}

TimerMenu ACCELERATORS
{
  VK_F2, IDM_DIALOG, VIRTKEY
  VK_F1, IDM_HELP, VIRTKEY
  "^X", IDM_EXIT
}
```

```
MYDB DIALOGEX 18, 18, 144, 100
CAPTION "A Countdown Timer"
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
{
  PUSHBUTTON "Start", IDD_START, 30, 80, 30, 14
  PUSHBUTTON "Cancel", IDCANCEL, 70, 80, 30, 14
  EDITTEXT IDD_EB1, 1, 1, 20, 12, ES_LEFT | WS_CHILD |
           WS_VISIBLE | WS_BORDER
  EDITTEXT IDD_EB2, 80, 1, 12, 12, ES_LEFT | WS_CHILD |
           WS_VISIBLE | WS_BORDER
  AUTOCHECKBOX "Show Countdown", IDD_CB1, 1, 48, 70, 10
  AUTOCHECKBOX "Beep At End", IDD_CB2, 1, 58, 60, 10
  AUTORADIOBUTTON "Minimize", IDD_RB1, 80, 48, 50, 10
  AUTORADIOBUTTON "Maximize", IDD_RB2, 80, 58, 50, 10
  AUTORADIOBUTTON "As-Is", IDD_RB3, 80, 68, 50, 10
}
```

머리부파일 TIMER.H의 내용은 다음과 같다.

```
#define IDM_DIALOG          100
#define IDM_EXIT            101
#define IDM_HELP            102

#define IDD_START           300
#define IDD_TIMER           301

#define IDD_CB1             400
#define IDD_CB2             401
#define IDD_RB1             402
#define IDD_RB2             403
#define IDD_RB3             404

#define IDD_EB1             500
#define IDD_EB2             501

#define IDD_UPDOWN          602
```

프로그람의 실행결과를 그림 11-1에 보여 주었다.

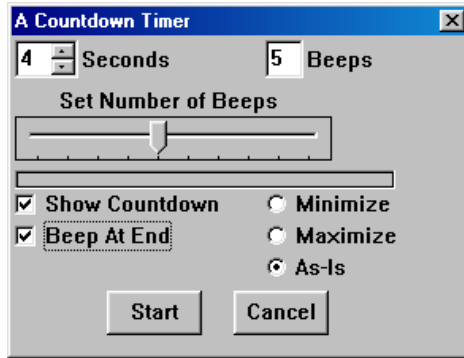



그림 11-1. 확장판시계프로그람의 실행결과

시계를 개시할 때([Start]단추를 누른다.) 돌리개조종체로부터 시계의 시간간격을 얻는다. [Show Countdown]에 검사표식이 붙어 있는 경우에는 시간이 경과할 때마다 돌리개조종체에 표시되는 남은 시간이 갱신된다.

시계의 시간간격이 경과하면 지정된 회수만큼 경보음이 울린다.([Beep At End]가 선택되여 있는 경우) 또한 경보음이 울릴 때마다 추적띠의 위치가 왼쪽으로 이동한다. 확장판 내려세기시계프로그람의 내용을 상세히 보자.

다시 한보 전진

Sleep( )함수

내려세기시계프로그람에서는 시계의 설정시간이 경과했을 때 경보음을 련속적으로 울리게 하기 위해 *Sleep( )*함수를 사용하고 있다. 경보음의 시간간격이 짧으면 끊기지 않는 하나의 련속적인 음으로 들리게 된다. 이러한 시간간격을 얻기 위해 쏘프트웨어에 의한 지연순환을 사용하였다. 아래에 그 실례를 보여 주었다.

```
for(x=0; x<100000; x++); // 단순한 지연순환
```

그러나 이러한 쏘프트웨어적인 수법에는 두가지 문제점이 있다. 첫번째 문제점은 쏘프트웨어의 지연순환이 시간적으로 정확치 않다는것이다. 그것은 지연시간이 CPU 의 속도에 따라 차이나기때문이다. 속도가 빠른 콤퓨터에서는 지연시간이 짧게 되고 속도가 뜬 콤퓨터에서는 지연시간이 길게 된다.

또한 CPU 의 속도가 같다고 하여도 사용하는 번역프로그람이 다르면 순환명령을 서로 다른 기계어코드로 변환하게 된다. 이것도 지연시간을 부정확하게 하는 원인으로 된다.

두번째 문제점은 쏘프트웨어의 지연순환이 CPU 에 공연한 부하를 준다는것

이다. 지연순환안에서 아무런 처리도 하지 않고 있다고 해도 체계에서 실행되고 있는 다른 프로세스의 CPU시간을 랑비하게 된다.

이 두가지 문제점을 극복하기 위해서 Windows 2000 에서는 *Sleep( )*함수를 사용하여 지연시간을 얻도록 해야 한다. Sleep( )함수는 ㎳단위로 지정된 시간만큼 프로그람의 실행을 정지시킨다. 아래에 프로그람코드를 보여 주었다.

```
VOID Sleep(DWORD delay);
```

delay 에 Sleep( )함수를 호출하고 있는 스레드를 정지시킬 시간을 ㎳단위로 설정한다. 프로그람이 실행중지상태로 되여 있는 동안에 CPU 시간은 체계에서 실행되고 있는 다른 프로세스에 할당된다.

돌리개조종체,추적띠 및 진행띠의 작성

돌리개조종체, *추적띠* 및 *진행띠*는 *DialogFunc( )*가 *WM_INITDIALOG* 통보문을 받았을 때 작성된다. 아래에 프로그람코드를 보여 주었다.

```
case WM_INITDIALOG:
     hEboxWnd = GetDlgItem(hdwnd, IDD_EB1);
     // 오르내리기조종체를 작성한다.
     udwnd = CreateUpDownControl(
               WS_CHILD | WS_BORDER | WS_VISIBLE |
               UDS_SETBUDDYINT | UDS_ALIGNRIGHT,
               10, 10, 50, 50,
               hdwnd,
               IDD_UPDOWN,
               hInst,
               hEboxWnd,
               MAXTIME, 1, udpos);
// 추적띠를 작성한다.
hTrackWnd = CreateWindow(TRACKBAR_CLASS,
               "",
               WS_CHILD | WS_VISIBLE | WS_TABSTOP |
               TBS_AUTOTICKS | WS_BORDER,
               2, 50,
               200, 28,
```

```
                hdwnd,
                NULL,
                hInst,
                NULL
};
SendMessage(hTrackWnd, TBM_SETRANGE,
                1, MAKELONG(low, high));
SendMessage(hTrackWnd, TBM_SETPOS,
                1, trackpos);
// 진행띠를 작성한다.
hProgWnd = CreateWindow(PROGRESS_CLASS,
                "",
                WS_CHILD | WS_VISIBLE | WS_BORDER,
                2, 84,
                240, 12,
                hdwnd,
                NULL,
                hInst,
                NULL);
// 증가걸음을 1로 한다.
SendMessage(hProgWnd, PBM_SETSTEP, 1, 0);

// [Beeps]편집칸에 수값을 설정한다.
SetDlgItemInt(hdwnd, IDD_KB2, trackpos, 1);
return 1;
```

우선 *CreateUpDownControl( )*을 호출하는 부분을 보자. 여기에서는 대화칸의 (10,10) 위치에 돌리개조종체를 작성하고 있다. 조종체의 너비는 50화소이며 높이도 50화소이다. 조종체는 대화칸의 새끼창문으로 되므로 어미창문의 손잡이로서 대화칸의 손잡이(hdwnd)가 설정되여 있다.

오르내리기조종체의 ID는 IDD_UPDOWN이다. hInst는 프로그람실체의 손잡이이다. 련동조종체는 hEboxWnd에서 지정되고 있다. 이것은 오르내리기조종체에 결합되는 편집칸의 손잡이이다. 오르내리기조종체의 범위는 1～MAXTIME이며 초기값은 1로 되여 있다.

돌리개조종체와 결합되는 편집칸이 자원파일에 정의되여 있으므로 그것의 손잡이를 얻기 위해 GetDlgItem( )을 호출하여야 한다.(제5장에서 GetDlgItem( )에 대하여 설명하고 있다.)

편집칸의 손잡이를 얻으면 그것을 GetUpDownControl( )함수의 련동조종체를 가리키는 파라메터에 설정한다. 편집칸을 련동조종체로 하여 오르내리기조종체를 작성하면 두개의 조종체가 자동적으로 결합되여 돌리개조종체로 된다.

다음 추적띠를 작성하고 있는 부분을 보자. 추적띠의 위치는 (2,50)이며 크기는 너비가 28화소, 높이 28화소로 되여 있다. 추적띠의 손잡이가 hTrackWnd에 보관된다. 추적띠를 작성한 다음 그의 범위를 1～BEEPMAX 로 설정하고 초기위치를 1 로 설정한다. *MAKELONG( )*마크로를 사용하여 범위를 설정하고 있다는데 주의를 돌려야 한다. 이 마크로는 두개의 옹근수값을 하나의 긴 옹근수로 결합한다. 마크로의 선언은 다음과 같다.

```
DWORD MAKELONG(WORD low, WORD high);
```

두배단어의 아래값을 low 에 설정하고 웃값을 high 에 설정한다. MAKELONG( )은 두 단어의 값을 긴 옹근수값으로 변환하는 경우에 편리하다.

마지막으로 진행띠를 작성하고 있는 부분을 보자. 진행띠는 추적띠의 가까이에 배치되고 걸음수에 1 을 설정하고 있다. 시계의 설정시간이 경과할 때마다 진행띠가 갱신되여 현재의 내려세기상황을 표시한다.

돌리개조종체의 조종

*돌리개조종체*가 조작되면 대화칸에 WM_VSCROLL 통보문을 보낸다. 이때는 오르내리기조종체의 손잡이가 lParam 에 보관되여 있다. 이 손잡이의 값과 CreateUpDownControl( )에서 돌려 진 값을 비교하여 통보문이 실제로 오르내리기조종체에서 보내는가를 검사한다.

이 실례프로그람에서는 같은 통보문을 보내는 조종체는 없으나 실제의 응용프로그람들에서는 여러개의 조종체에서 WM_VSCROLL 통보문을 보낼 가능성도 없지 않다. 그러므로 통보문을 보낸 조종체를 항상 검사해야 한다.

돌리개조종체의 새로운 위치는 HIWORD(wParam)에 보관되여 있다. 이 값이 10의 배수일 때는 경보음을 울린다. 사용자가 시간간격을 증가하거나 감소시키면 그 값이 10의 배수로 될 때마다 경보음이 울린다. 이 기능은 긴 지연시간을 설정하는 경우에 편리할것이다.

돌리개조종체의 조종은 완전히 자동화되여 있으므로 WM_VSCROLL 통보문의 처리밖에는 아무것도 할 필요가 없다. 10 의 배수에서 경보음을 울리게 하지 않는다면 WM_VSCROLL 통보문을 처리할 필요도 없다. 사실 돌리개조종체를 사용하는 대부분의 프로그람들에서는 현재의 설정값을 얻는것외에 조종체를 조종하는 경우가 없다.

IDD_START 를 처리하고 있는 부분에서는 UDM_GETPOS 를 돌리개조종체의 현재의 값을 얻고 있다. 이 통보문을 받으면 돌리개조종체는 련동조종체에 표시되여 있는 값을 돌려 준다. 편집칸의 값이 항상 돌리개조종체의 현재의 값을 표시하고 있다는데 주의를 돌려야 한다. 이것은 사용자가 수동적으로 새로운 값을 입력한 경우에도 같다.

추적띠의 조종

추적띠가 이동되면 WM_HSCROLL 통보문을 보내여 아래와 같은 처리가 진행된다.

```
case WM_HSCROLL: // 추적띠가 능동
      if(hTrackWnd != (HWND)lParam) break; // 추적띠가 아닌 경우

      switch(LOWORD(wParam)) {
        case TB_TOP:
        case TB_BOTTOM:                 // 이 실례프로그람에서는
        case TB_LINEUP:                 // 모든 통보문을
        case TB_LINEDOWN:               // 같은 방법으로
        case TB_THUMBPOSITION:          // 처리한다.
        case TB_THUMBTRACK:
        case TB_PAGEUP:
        case TB_PAGEDOWN:
          trackpos = SendMessage(hTrackWnd, TBM_GETPOS,
                        0, 0);
          SetDlgItemInt(hdwnd, IDD_EB2, trackpos, 1);
          return 1;
      }
      break;
```

사용자가 추적띠의 조절기를 움직이면 추적띠의 위치가 자동적으로 갱신된다. 따라서 그 처리를 프로그람에서 진행할 필요는 없다. 추적띠의 동작이 끝나면 프로그람에서는 새로운 위치를 얻고 그 값을 경보음의 회수를 가리키는 편집칸에 설정한다.

추적띠의 현재값은 *SetDlgItemInt( )*을 사용하여 [Beeps]편집칸에 표시된다. 이 함수는 편집칸에 옹근수값을 표시한다. 선언은 다음과 같다.

```
BOOL SetDlgItemInt(HWND hDialog, int ID, UINT value, BOOL
        signed);
```

hDialog 에는 편집칸을 포함하고 있는 대화칸의 손잡이를 설정하고 ID 에는 편집칸의 ID 를 설정한다. value 에 편집칸에 표시시킬 값을 설정한다. signed 에 령이 아닌 값을 설정한 경우는 value 에 부의 값을 설정할수 있게 된다. 그렇지 않은 경우에는 값이 부호가 없는것으로서 인정된다. 이 함수는 호출이 성공하면 령 아닌 값을 돌려 주며 실패하면 령을 돌려 준다.

사용자가 [Beeps]편집칸을 사용하여 수동적으로 경보회수를 설정할수도 있다. 이 경우에는 사용자가 입력한 값이 다음의 프로그람코드에서 추적띠에 반영된다.

```
case IDD_EB2:
    /* 사용자가 [Beeps]편집칸에 수값을
       입력하면 추적띠를 갱신한다. */
    trackpos = GetDlgItemInt(hdwnd, IDD_EB2, NULL, 1);
    SendMessage(hTrackWnd, TBM_SETPOS, 1, trackpos);
    return 1;
```

편집칸안의 값이 *GetDlgItemInt( )*에 의해 얻어 진다는데 주의를 돌려야 한다. 이 함수는 편집칸에 현재 표시된 문자렬을 옹근수값으로 변환하여 돌려 준다.

실례로 편집칸에 102 라는 문자렬이 표시되여 있다면 *GetDlgItemInt( )*의 돌림값은 102 라는 옹근수값으로 된다. 여러가지 리유로부터 이 기능은 수자를 표시하는 편집칸에만 적용된다. *GetDlgItemInt( )*함수의 선언은 다음과 같다.

```
UINT GetDlgItemInt(HWND hDialog, int ID, BOOL *error, BOOL signed);
```

편집칸을 포함하고 있는 대화칸의 손잡이를 hDialog 에 설정한다. ID 에는 편집칸의 ID 를 설정한다. 편집칸에 수값으로 변환할수 있는 문자렬이 표시되여 있지 않는 경우에는 령이 돌려 진다. 그러나 령도 하나의 수값이므로 함수의 호출이 성공하였는가 아닌가는 error 에 돌려 지게 되여 있다.

함수를 호출하여 error 에 령 아닌 값이 돌려 졌다면 함수의 돌림값은 유효한것이다. 오유가 발생한 경우에는 error 에 령이 돌려 진다. 오유를 무시해도 상관이 없다면 이 파라메터에 NULL 을 설정해야 한다. signed 에 령 아닌 값을 설정한 경우는 *GetDlgItemInt( )*가 부호 있는 값을 돌려 준다. 그렇지 않은 경우에는 부호 없는 값이 돌려 진다.

이 실례프로그람은 추적띠를 마우스나 건반의 임의의것으로 동작시킬수 있게 되여 있다. 프로그람에서 많은 TB_통보문을 처리하고 있는것은 건반입력을 지원하고 있기때문이다. 이 통보문들을 몇개 삭제하여 그 결과를 확인해 볼수도 있다.

진행띠의 조종

*진행띠*는 내려세기의 진행상황을 표시한다. 내려세기의 진행상황을 정확히 표시하려면 진행띠범위의 최대값을 내려세기의 초수에 맞추어야 한다. 진행띠의 초기값은 령으로 해야 한다.

이러한 처리들은 IDD_START 의 case 문에서 진행되는데 이것은 사용자가 [Start] 단추를 눌렀을 때 실행된다. 아래에 프로그람코드를 보여 주었다.

```
// 진행띠를 초기화한다.
```

```
SendMessage(hProgWnd, PBM_SETRANGE, 0,
            MAKELONG(0, udpos);
SendMessage(hProgWnd, PBM_SETRANGE, 0, 0);
```

udpos 에는 돌리개조종체에서 설정된 초수가 보관되여 있다. 시계의 설정시간이 경과하면 아래의 프로그람코드에서 진행띠가 갱신된다.

```
//진행띠를 전진시킨다.
SendMessage(hProgWnd, PBM_STEPIT, 0, 0);
```

이리하여 진행띠의 위치는 한걸음 앞으로 전진한다.

간단한 코드로 고급한 기능의 실현

앞에서 본것처럼 아주 간단한 코드만으로 오르내리기조종체, 추적띠 및 진행띠를 프로그람에 받아 들일수 있었다. 그러나 그 결과는 매우 큰것이다. 제 6 장에서 작성한 내려세기시계프로그람과 이 장에서 작성한 확장용 내려세기시계프로그람의 모양, 조작기능을 비교해 보면 어느 프로그람이 보다 시각적효과가 큰 대면부를 가지고 있는가를 명백히 알수 있다.

다시 한보 전진

추적띠에서 련동조종체를 사용하는 방법

이미 설명한것처럼 최근에 와서 추적띠에 *련동조종체*를 결합시키는 기능이 추가되였다. 오르내리기조종체의 경우와 같이 추적띠의 련동조종체는 일반적으로 *편집칸*으로 된다.

추적띠에 련동조종체를 결합하려면 추적띠를 작성한후에 TBM_SETBUDDY 통보문을 보내야 한다. 이때 lParam 에 련동조종체의 손잡이를 설정하고 wParam 에는 련동조종체의 위치를 설정한다.

수평추적띠의 왼쪽 또는 수직추적띠의 밑에 련동조종체를 배치하려는 경우에는 wParam 에 령을 설정한다. 수평추적띠의 왼쪽 또는 수직흘림띠의 오른쪽에 련동조종체를 배치하는 경우에는 wParam 에 령 아닌 값을 설정한다.

추적띠는 하나 혹은 두개의 련동조종체를 가질수 있으며 련동조종체를 두개 가지는 경우에는 그것들이 추적띠의 량쪽에 표시된다.

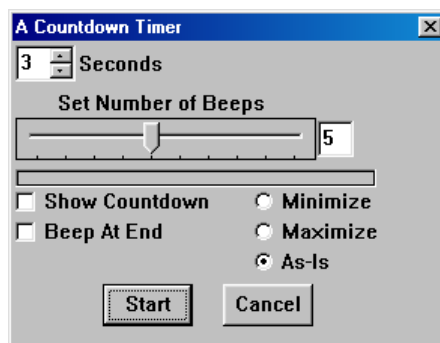
이 장의 실레프로그람을 개조하여 [Beeps]편집칸을 추적띠의 련동조종체로

만들어 보자. 우선 WM_PAINT 통보문을 처리하는 case 문에서 아래의 부분은 필요 없으므로 삭제한다.

```
sprintf(str, "Beeps");
TextOut(hdc, 186, 6, str, strlen(str));
```

다음은 추적띠를 작성한후에 *TBM_SETBUDDY* 통보문을 보내도록 한다.

```
SendMessage(hTrackWnd, TBM_SETBUDDY, 0, (LPARAM)
            GetDlgItem(hdwnd, IDD_EB2));
```

GetDlgItem( )은 돌림값으로서 [Beeps]편집칸의 손잡이를 돌려 주므로 그것을 추적띠의 련동조종체로서 설정한다. 이러한 변경을 진행하면 추적띠의 외형은 아래와 같이 변화된다.

제 12 장

상태창문, 표쪽조종체 및 나무구조보기조종체

이 장에서는 계속하여 Windows 2000 의 공통조종체들인 상태창문, 표쪽조종체 및 나무구조보기조종체들에 대하여 설명한다.

상태창문의 사용방법

프로그람의 상태나 속성 또는 어떤 파라메터 등의 정보를 항상 표시해 두고 싶은 경우가 흔히 있다. Windows 2000 은 이러한 요구를 만족시켜 주기 위하여 상태창문(Status window) 또는 *상태띠*(Status bar)라고 불리우는 조종체를 제공하고 있다.

*상태창문*은 일반적으로 창문의 밑부분에 표시되는 띠로 되여 있다. 이 띠에 프로그람과 관계되는 정보가 표시된다.

프로그람에 상태창문을 추가하는것은 아주 간단하다. 모든 응용프로그람들에서 상태창문을 리용하면 통일적인 형식으로 상태정보를 표시할수 있다.

상태창문의 작성

상태창문을 작성하려면 *CreateStatusWindow( )*함수를 사용한다. 아래에 선언을 보여 주었다.

```
HWND CreateStatusWindow(LONG WinStyle, LPCSTR lpszFirstPart,
                        HWND hParent, UINT ID);
```

WinStyle 에 상태창문의 형식을 설정한다. 이 형식에는 *WM_CHILD* 를 포함시켜야 한다. 또한 일반적으로 상태창문이 자동적으로 표시되게 WS_VISIBLE 도 포함시킨다.

상태창문을 두개이상의 령역으로 갈라서 매 령역에 본문을 표시할수도 있다. 상태창문의 첫 령역에 표시할 문자렬의 지시자를 lpszFirstPart 에 설정한다. 이 문자렬을 후에 설정한다면 lpszFirstPart 에 NULL 을 설정한다.

어미창문의 손잡이를 hParent 에 설정하고 상태창문의 식별자를 ID 에 설정한다. 이 함수는 상태창문의 손잡이를 돌려 준다. 함수의 호출이 실패한 경우에는 NULL 이 돌려진다.

상태창문을 작성할 때는 그 크기나 위치를 설정하지 않는다는 점에 주의해야 한다. 그 리유는 상태창문이 자동적으로 어미창문의 아래부분에 적절한 크기로 배치되기때문이다. 만일 어미창문의 웃부분에 상태창문을 배치하려고 한다면 WinStyle 파라메터에 *CCS_TOP*를 포함시킨다. *SBARS_TOOLTIPS*(낡은 방식의 마크로인 SBT_TOOLTIPS 를 사용할수도 있다.)를 포함시키면 도구설명쪽지를 표시할수도 있다.

참고 *: 창문클라스에 STATUSCLASSNAME 을 지정하고 CreateWindow( ) 혹은 CreateWindowEx( )의 어느 하나를 사용하여 상태창문을 작성할수도 있다. 그러나 CreateStatusWindow( )를 사용하는 편이 보다 간단하다.*

상태창문은 두개이상의 령역으로 갈라서 리용하는것이 일반적이다.(그러나 령역이 한개라고 해도 전혀 문제가 없다.) 령역을 가르면 매개 령역에 서로 다른 본문을 표시할수 있다. 매 령역은 색인에 의해서 참조된다. 선두령역의 색인은 령으로 된다. 이미 설명한것처럼 상태창문을 작성할 때 선두령역에 표시할 본문을 설정할수 있다.

상태창문의 통보문

상태창문은 통보문을 생성하지 않지만 SendMessage( )를 사용하여 프로그람으로부터 상태창문에 통보문을 보낼수 있다. 상태창문에 보낼수 있는 주요한 통보문들을 표 12-1 에 주었다.

대부분의 응용프로그람들에서는 SB_SETPARTS 및 SB_SETTEXT 통보문을 상태창문에 보낸다. 이 통보문들은 상태창문의 령역의 수와 매 령역에 표시할 본문을 설정한다. 아래에 상태창문을 작성하기 위한 일반적인 순서를 보여 주었다.

- 상태창문을 작성한다.
- SB_SETPARTS 통보문을 보내여 령역의 수를 설정한다.
- SB_SETTEXT 통보문을 보내여 매 령역에 본문을 설정한다.

상태창문의 초기화가 끝나면 필요에 따라 SB_SETTEXT 통보문을 보내여 매 령역에 표시된 문자렬을 갱신할수 있다.

표 12-1. 상태창문의 주요한 통보문

통 보 문	의 미
SB_GETPARTS	상태창문령역의 오른쪽 끝의 X 자리표를 얻는다. 상태창문 령역의 수가 돌려 진다.(호출이 실패한 경우는 령이 돌려 진다.) wParam 에는 얻으려는 령역의 수를 설정한다. lParam 에는 매개 령역의 오른쪽 끝의 X 자리표를 보관하기 위한 옹근수형배렬의 지시자를 설정한다. 이 배렬의 크기는 적어도 요구된 령역의 수이상이여야 한다. 창문의 경계선에 접하고 있는 경우에는 X 자리표로서 -1 이 돌려진다.

SB_GETTEXT	지정한 령역의 본문을 얻는다. 돌림값의 아래단어에 본문의 문자수가 보관된다.아래단어에는 본문의 표시방법을 지적하는 값이 보관된다. 이 값이 령이면 본문이 오목패이게 표시되여 있다. SBT_POPOUT 라면 본문이 도두라지게 표시된다.SBT_NOBORDERS 라면 본문이 경계선이 없이 표시된다. SBT_RTLREADING 이라면 본문이 오른쪽에서 왼쪽으로 표시된다. wParam 에는 목적하는 령역의 색인을 설정한다. lParam 에는 본문을 보관할 문자렬의 지시자를 설정한다.(이 배렬의 크기는 지정된 령역에 표시되여 있는 본문을 보관하는데 충분한 크기로 되여야 한다.)
SB_SETPARTS	상태창문령역의 수를 설정한다. 호출이 성공하면 령 아닌 값을 돌려 주며 실패하면 령을 돌려 준다.wParam 에는 령역의 수를 설정한다. lParam 에는 매개 령역의 오른쪽 끝의 X 자리표를 보관하고 있는 옹근수형배렬의 지시자를 설정한다. 어미창문의 오른쪽 끝의 경계선을 지정하는 경우에는 -1 을 설정한다.
SB_SETTEXT	령역에 표시할 문자렬을 설정한다. 호출이 성공하면 령 아닌 값이 돌려 지고 실패하면 령을 돌려 준다. wParam 에는 본문을 설정하는 령역의 색인과 본문의 표시방법을 지정하는 값을 OR 연산하여 설정한다. 이 값이 령이면 본문이 오목패이게 표시된다.(체계설정) SBT_POPOUT 라면 본문이 도드라지게 표시된다. SBT_NOBORDERS 라면 본문이 경계선이 없이 표시된다. SBT_OWNERDRAW 라면 어미창문이 본문을 표시한다. SBT_RTLREADING 이라면 본문이 오른쪽에서 왼쪽으로 표시된다. lParam 에는 표시할 문자렬의 지시자를 설정한다.
SB_SETTIPTEXT	도구설명쪽지본문을 설정한다. wParam 에는 도구설명쪽지본문을 표시할 령역의 색인을 설정한다. lParam 에는 표시할 본문을 보관한 문자렬의 지시자를 설정한다.

상태창문의 사용방법

실례 12-1 의 프로그람은 *대화칸*안의 설정정보를 표시하기 위해 상태창문을 사용하는 프로그람이다. 이 대화칸에는 *돌리개조종체*와 세개의 *검사칸* 및 두개의 *단일선택단추*가 있다. *상태창문*에는 이 조종체들의 현재설정상태가 표시된다.

이 프로그람을 번역할 때는 COMCTL32.LIB 를 련결해야 한다. 프로그람의 실행결과를 그림 12-1 에 보여 주었다.

실례 12-1. Status 프로그람

```
// 상태창문의 실례

#include <windows.h>
#include <commctrl.h>
#include <cstdio>
#include "status.h"

#define NUMPARTS 5

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK DialogFunc(HWND, UINT, WPARAM, LPARAM);
void InitStatus(HWND hwnd);

char szWinName[] = "MyWin"; // 창문클라스의 이름

HINSTANCE hInst;
HWND hwnd;
HWND hStatusWnd;

int parts[NUMPARTS];

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  MSG msg;
  WNDCLASSEX wcl;
  HACCEL hAccel;
  INITCOMMONCONTROLSEX cc;

  // 창문클라스를 정의한다.
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst;      // 실체의 손잡이
  wcl.lpszClassName = szWinName; // 창문클라스의 이름
  wcl.lpfnWndProc = WindowFunc;  // 창문함수
  wcl.style = 0;                  // 체계설정의 형식
```

```
wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

wcl.lpszMenuName = "StatusMenu"; // 기본차림표

wcl.cbClsExtra = 0; // 보조기억기령역은 필요 없다.
wcl.cbWndExtra = 0;

// 창문의 배경색을 흰색으로 한다.
wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

// 창문클라스를 등록한다.
if(!RegisterClassEx(&wcl)) return 0;

/* 창문클라스가 등록되였으므로
   창문을 작성할수 있다. */
hwnd = CreateWindow(
  szWinName, // 창문클라스의 이름
  "Using a Status Bar", // 제목
  WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
  CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 너비는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 높이는 Windows가 결정하게 한다.
  NULL,           // 어미창문은 없다.
  NULL,           // 차림표는 없다.
  hThisInst,      // 실체의 손잡이
  NULL            // 추가파라메터는 없다.
);

hInst = hThisInst; // 현재실체의 손잡이를 보관한다.

// 건반가속기를 적재한다.
hAccel = LoadAccelerators(hThisInst, "StatusMenu");
// 공통조종체를 초기화한다.
cc.dwSize = sizeof(INITCOMMONCONTROLSEX);
```

```
    cc.dwICC = ICC_BAR_CLASSES | ICC_UPDOWN_CLASS;
    InitCommonControlsEx(&cc);

    // 창문을 표시한다.
    ShowWindow(hwnd, nWinMode);
    UpdateWindow(hwnd);

    // 통보문순환고리를 작성한다.
    while(GetMessage(&msg, NULL, 0, 0))
    {
      if(!TranslateAccelerator(hwnd, hAccel, &msg)) {
        TranslateMessage(&msg); // 건반통보를 변환한다.
        DispatchMessage(&msg); // Windows 2000 에 조종을 넘긴다.
      }
    }

    return msg.wParam;
  }

  /* 이 함수는 Windows 2000 으로부터 호출되여
     통보문대기렬에서 꺼낸 통보문을 받아 들인다.
  */
  LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                              WPARAM wParam, LPARAM lParam)
  {
    int response;

    switch(message) {
      case WM_COMMAND:
        switch(LOWORD(wParam)) {
          case IDM_DIALOG:
            DialogBox(hInst, "StatusDB", hwnd, (DLGPROC) DialogFunc);
            break;
          case IDM_EXIT:
            response = MessageBox(hwnd, "Quit the Program?",
                                  "Exit", MB_YESNO);
            if(response == IDYES) PostQuitMessage(0);
            break;
```

```
        case IDM_HELP:
          MessageBox(hwnd, "Try the Dialog Box",
                     "Help", MB_OK);
          break;
      }
      break;
    case WM_DESTROY: // 프로그람을 끝낸다.
      PostQuitMessage(0);
      break;
    default:
      /* 이 switch 문에서 지정된것 이외의 통보문은
         Windows 2000 에 처리를 맡긴다. */
      return DefWindowProc(hwnd, message, wParam, lParam);
  }

  return 0;
}

// 상태창문의 실례를 위한 대화함수
BOOL CALLBACK DialogFunc(HWND hdwnd, UINT message,
                         WPARAM wParam, LPARAM lParam)
{
  static long udpos = 0;
  static char str[80];
  static HWND hEboxWnd;
  static HWND udWnd;
  static statusCB1, statusCB2;
  static statusRB1;
  int low=0, high=10;

  switch(message) {
    case WM_INITDIALOG:
      InitStatus(hdwnd); // 상태창문을 초기화한다.

      hEboxWnd = GetDlgItem(hdwnd, ID_EB1);
      udWnd = CreateUpDownControl(
                 WS_CHILD | WS_BORDER | WS_VISIBLE |
                 UDS_SETBUDDYINT | UDS_ALIGNRIGHT,
```

```
                10, 10, 50, 50,
                hdwnd,
                ID_UPDOWN,
                hInst,
                hEboxWnd,
                high, low, high/2);

       // 단일선택단추를 초기화한다.
       SendDlgItemMessage(hdwnd, ID_RB2, BM_SETCHECK,
                          BST_CHECKED, 0);
       return 1;
     case WM_VSCROLL: // 오르내리기조종체를 처리한다.
       if(udWnd==(HWND)lParam) {
         udpos = GetDlgItemInt(hdwnd, ID_EB1, NULL, 1);
         sprintf(str, "Power: %d", udpos);
         SendMessage(hStatusWnd, SB_SETTEXT,
                     (WPARAM) 0, (LPARAM) str);
       }
       return 1;
     case WM_COMMAND:
       switch(LOWORD(wParam)) {
         case ID_CB1: // 「Optimize」검사칸을 처리한다.
           statusCB1 = SendDlgItemMessage(hdwnd, ID_CB1,
                       BM_GETCHECK, 0, 0);
           if(statusCB1) sprintf(str, "Optimize: Yes");
           else sprintf(str, "Optimize: No");
           SendMessage(hStatusWnd, SB_SETTEXT,
                     (WPARAM) 1, (LPARAM) str);
           return 1;
         case ID_CB2: // 「Debug」검사칸을 처리한다.
           statusCB2 = SendDlgItemMessage(hdwnd, ID_CB2,
                       BM_GETCHECK, 0, 0);
           if(statusCB2) sprintf(str, "Debug: Yes");
           else sprintf(str, "Debug: No");
           SendMessage(hStatusWnd, SB_SETTEXT,
                     (WPARAM) 2, (LPARAM) str);
           return 1;
         case ID_CB3: // 「NT 4」검사칸을 처리한다.
```

```
    statusCB2 = SendDlgItemMessage(hdwnd, ID_CB3,
                BM_GETCHECK, 0, 0);
    if(statusCB2) sprintf(str, "NT 4: Yes");
    else sprintf(str, "NT 4: No");
    SendMessage(hStatusWnd, SB_SETTEXT,
              (WPARAM) 3, (LPARAM) str);
    return 1;
  case ID_RB1: // 단일선택단추를 처리한다.
  case ID_RB2:
    statusRB1 = SendDlgItemMessage(hdwnd, ID_RB1,
                BM_GETCHECK, 0, 0);
    if(statusRB1) sprintf(str, "Using C");
    else sprintf(str, "Using C++");
    SendMessage(hStatusWnd, SB_SETTEXT,
                (WPARAM) 4, (LPARAM) str);
    return 1;
  case ID_RESET: // 추가선택항목을 재설정한다.
    SendMessage(udWnd, UDM_SETPOS, 0, (LPARAM) high /2);
    SendMessage(hStatusWnd, SB_SETTEXT, (WPARAM) 0,
        (LPARAM) "Power: 5");
    SendDlgItemMessage(hdwnd, ID_CB1,
                BM_SETCHECK, 0, 0);
    SendDlgItemMessage(hdwnd, ID_CB2,
                BM_SETCHECK, 0, 0);
    SendDlgItemMessage(hdwnd, ID_CB3,
                BM_SETCHECK, 0, 0);
    SendMessage(hStatusWnd, SB_SETTEXT, (WPARAM) 1,
        (LPARAM) "Optimize: No");
    SendMessage(hStatusWnd, SB_SETTEXT, (WPARAM) 2,
        (LPARAM) "Debug: No");
    SendMessage(hStatusWnd, SB_SETTEXT, (WPARAM) 3,
        (LPARAM) "NT 4: No");
    SendMessage(hStatusWnd, SB_SETTEXT, (WPARAM) 4,
        (LPARAM) "Using C++");
    return 1;
  case IDCANCEL:
  case IDOK:
    EndDialog(hdwnd, 0);
```

```
            return 1;
        }
    }
    return 0;
}

// 상태창문의 초기화
void InitStatus(HWND hwnd)
{
    RECT WinDim;
    int i;

    GetClientRect(hwnd, &WinDim);

    for(i=1; i<=NUMPARTS; i++)
        parts[i-1] = WinDim.right/NUMPARTS * i;

    // 상태창문을 작성한다.
    hStatusWnd = CreateStatusWindow(
                    WS_CHILD | WS_VISIBLE,
                    "Power: 5", hwnd,
                    ID_STATUSWIN);

    SendMessage(hStatusWnd, SB_SETPARTS,
                (WPARAM) NUMPARTS, (LPARAM) parts);

    SendMessage(hStatusWnd, SB_SETTEXT, (WPARAM) 1,
                (LPARAM) "Optimize: No");
    SendMessage(hStatusWnd, SB_SETTEXT, (WPARAM) 2,
                (LPARAM) "Debug: No");
    SendMessage(hStatusWnd, SB_SETTEXT, (WPARAM) 3,
                (LPARAM) "NT 4: No");
    SendMessage(hStatusWnd, SB_SETTEXT, (WPARAM) 4,
                (LPARAM) "Using C++");
}
```

이 프로그람은 아래의 자원파일을 필요로 한다.

```
#include <windows.h>
```

```
#include "status.h"

StatusMenu MENU
{
  POPUP "&Options" {
    MENUITEM "&Dialog\tF2", IDM_DIALOG
    MENUITEM "E&xit\tCtrl+X", IDM_EXIT
  }
  MENUITEM "&Help", IDM_HELP
}

StatusMenu ACCELERATORS
{
  VK_F1, IDM_HELP, VIRTKEY
  VK_F2, IDM_DIALOG, VIRTKEY
  "^X", IDM_EXIT
}

StatusDB DIALOGEX 18, 18, 180, 92
CAPTION "Demonstrate a Status Bar"
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
{
  PUSHBUTTON "Reset", ID_RESET, 112, 40, 37, 14
  PUSHBUTTON "OK", IDOK, 112, 60, 37, 14
  EDITTEXT ID_EB1, 10, 10, 20, 12, ES_LEFT |
           WS_BORDER | WS_TABSTOP
  LTEXT "Power Factor", ID_STEXT1, 36, 12, 50, 12
  AUTOCHECKBOX "Optimize", ID_CB1, 10, 35, 48, 12
  AUTOCHECKBOX "Debug Info", ID_CB2, 10, 50, 48, 12
  AUTOCHECKBOX "NT 4 Compatible", ID_CB3, 10, 65, 70, 12
  AUTORADIOBUTTON "C", ID_RB1, 112, 8, 20, 12
  AUTORADIOBUTTON "C++", ID_RB2, 112, 20, 28, 12
  LTEXT "Language", ID_STEXT2, 140, 14, 40, 12
}
```

머리부파일 STATUS.H의 내용을 아래에 보여 주었다.

```
#define IDM_DIALOG          100
```

```
#define IDM_EXIT            101
#define IDM_HELP            102
#define ID_UPDOWN           103
#define ID_EB1              104
#define ID_EB2              105
#define ID_CB1              106
#define ID_CB2              107
#define ID_CB3              108
#define ID_RB1              109
#define ID_RB2              110
#define ID_RESET            111

#define ID_STATUSWIN        200

#define ID_STEXT1           300
#define ID_STEXT2           301
```

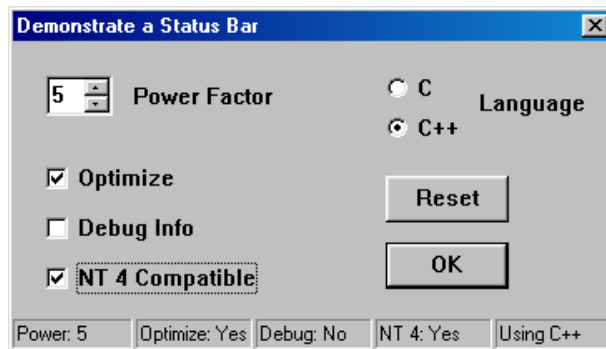



그림 12-1. 상태창문프로그람의 실행결과

이 프로그람에서는 InitStatus( )함수에서 상태창문의 작성과 초기화를 진행한다.

우선 *GetClientRect( )*를 사용하여 대화칸의 크기를 얻는다. 창문의 너비를 NUMPARTS(이 옹근수의 값은 5 이다.)로 나누어 상태창문의 매개 령역의 오른쪽 끝위치를 결정하고 그 값들을 배렬 parts 에 보관한다. 매 령역의 너비가 아니라 매 령역의 오른쪽 끝의 위치를 상태창문에 설정한다는 점에 주목해야 한다.

다음 상태창문을 작성하고 령역을 설정한다. 마감으로 매개 령역에 본문의 초기값을 설정한다.(선두령역의 본문은 CreateStatusWindow( )에서 설정한다.)

DialogFunc( )안에서는 조종체의 상태가 변경될 때마다 상태창문의 표시를 갱신하고 있다. 이것은 매 조종체에 대응되는 상태창문의 령역에 SB_SETTEXT 통보문을 보내여 실현된다.

상태창문의 령역에 표시되는 본문은 령역의 내부에 들어 맞도록 자동적으로 잘라 진다. 본문이 이웃한 령역을 침범하여 표시되는 일은 없다.

다시 한보 전진

상태창문의 크기변경

상태창문의 크기는 *어미창문*에 맞추어 초기화되지만 어미창문의 크기가 변하면 상태창문의 크기도 그에 맞게 자동적으로 변화되지는 않는다.

어미창문안에 있는 상태창문(새끼창문)의 크기를 변경하려면 SendMessage( )함수를 사용하여 어미창문으로부터 새끼창문에 *WM_SIZE*통보문을 보내야 한다.(이 처리는 제 10 장에서 설명한 도구띠의 크기변경과 류사하다.)

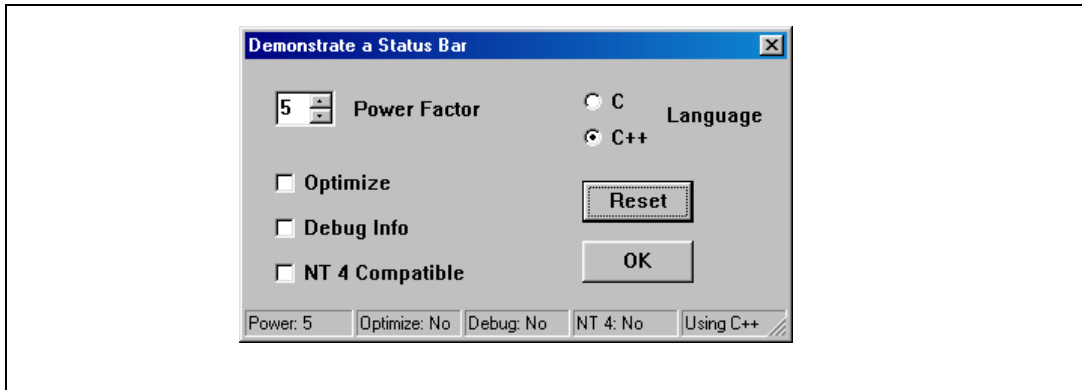
앞서 본 실례프로그람에서 어미창문인 대화칸의 크기가 변경될 때 자동적으로 상태창문의 크기가 변경되도록 하려면 DialogFunc( )에 아래의 case 문을 추가해야 한다.

```
case WM_SIZE:
  SendMessage(hStatusWnd, WM_SIZE, wParam, lParam);
  GetClientRect(hdwnd, &WinDim);
  for(i=1; i<=NUMPARTS; i++)
    parts[i-1] = WinDim.right/NUMPARTS * i;

SendMessage(hStatusWnd, SB_SETPARTS,
            (WPARAM) NUMPARTS, (LPARAM) parts);
  return 1;
```

DialogFunc( )에는 아래의 변수도 추가하여야 한다.

```
RECT WinDim;
int i;
```

또한 대화칸의 크기를 변경가능하게 하기 위해 대화칸의 자원에 WS_SIZEBOX형식을 포함시켜야 한다.

이러한 변경을 진행하고 상태창문의 프로그람을 실행한 다음 대화칸의 크기를 변경해 보면 상태창문의 크기도 자동적으로 변경된다는것을 알수 있다. 크기를 변경한 대화칸을 아래에 보여 주었다.

표쪽조종체의 소개

표쪽조종체(tab control)는 시각적효과가 큰 공통조종체의 하나이다. 표쪽조종체는 여러개 서류철등록부의 표쪽들을 줄 지어 놓은것이다. 표쪽이 선택되면 그에 대응하는 등록부가 전면에 표시된다.

표쪽조종체의 사용방법은 간단하지만 그것을 실현하는 프로그람의 작성은 약간 복잡하다. 여기에서는 표쪽조종체의 기본적인 기능에 대해 설명한다. 다음 절에서는 표쪽조종체의 고급한 기능을 설명한다.

표쪽조종체의 작성

표쪽조종체를 작성하려면 창문클라스에 *WC_TABCONTROL* 을 지정하고 CreateWindow( ) 또는 CreateWindowEx( )를 사용한다. 표쪽조종체는 새끼창문(*WS_CHILD*)으로 된다. 또한 창문의 형식에 WS_VISIBLE 을 지정하여 표쪽조종체가 자동적으로 표시되게 하는것이 일반적이다. 표쪽조종체를 작성하기 위한 프로그람코드의 실례를 아래에 보여 주었다.

```
hTabWnd = CreateWindow(
                WC_TABCONTROL,
                "",
                WS_VISIBLE | WS_TABSTOP | WS_CHILD,
                0, 0, 100, 100,
                hwnd,   // 어미창문의 손잡이
                NULL,
                hInst,  // 실체손잡이
                NULL
```

```
    };
```

표쪽조종체에 설정할수 있는 형식에는 여러가지가 있다. 실례로 *TCS_BUTTONS* 는 표쪽을 단추형태로 한다. 그러나 대부분의 응용프로그람에서는 체계설정형식의 표쪽이면 충분하다. 표쪽조종체가 작성되면 응용프로그람으로부터 표쪽조종체에 통보문을 보낼수 있다. 표쪽조종체가 조작되면 표쪽조종체가 통보문을 생성한다.

작성된 직후의 표쪽조종체의 내부는 비여 있다. 표쪽조종체를 사용하기에 앞서 표쪽을 삽입하여야 한다. 매개 표쪽은 아래에 보여 준 *TCITEM* 구조체에서 정의된다.

```
typedef struct tagTCITEM
{
    UINT mask;
    DWORD dwState;
    DWORD dwStateMask;
    LPSTR pszText;
    int cchTextMax;
    int iImage;
    LPARAM lParam;
}   TCITEM;
```

TCITEM 구조체에서 mask 의 값은 dwState, pszText 및 iImage 등의 성원에 유효한 값이 보관되여 있는가 아닌가를 가리키게 된다. mask 의 값은 아래에 보여 준 몇가지 값들의 조합으로 된다.

mask 의 값	의　　미
TCIF_IMAGE	iImage 에 자료가 보관되여 있다.
TCIF_PARAM	lParam 에 자료가 보관되여 있다.
TCIF_STATE	dwState 에 자료가 보관되여 있다.
TCIF_TEXT	pszText 에 자료가 보관되여 있다.

mask 에 TCIF_RTLREADING 라는 값을 포함시킬수도 있다. 이것은 본문이 오른쪽에서 왼쪽으로 표시된다는것을 가리킨다.

표쪽을 작성할 때는 dwState 의 값이 사용되지 않는다. 현재 존재하는 표쪽의 상태를 얻는 때 dwState 에 정보가 보관된다. dwState 의 값은 령, TCIS_BUTTONPRESSED(단추형태의 표쪽에서만 사용) 혹은 TCIS_HIGHLIGHTED(표쪽이 강조표시되고 있다.)의 어느 한 값으로 된다.

dwStateMask 의 값은 dwState 의 유효비트를 지적한다. dwStateMask 는 항목을

삽입할 때는 사용되지 않는다.

표쪽을 설정할 때는 pszText 에 표쪽안에 표시될 문자렬의 지시자를 설정한다. 표쪽의 정보를 얻을 때는 pszText 에 본문을 보관할 배렬의 지시자를 설정한다. 이 경우에는 pszText 에서 지적되는 배렬의 크기를 cchTextMax 에 설정한다.

표쪽조종체에 화상목록을 련관시키는 경우에는 특정한 표쪽에 련관시킬 화상의 색인을 iImage 에 설정한다. 표쪽조종체에 련관시킬 화상목록이 없는 경우에는 iImage 에 -1 을 설정한다. 이 장의 실례프로그람에서는 화상을 사용하지 않는다.

lParam 에는 응용프로그람자체의 자료를 보관할수 있다.

이식과 관련한 요점 : TCITEM 구조체는 낡은 형인 TC_ITEM 구조체를 치환한것이다. TC_ITEM 구조체에서는 dwState 와 dwStateMask 가 미리 예약된 성원으로 되여 있다.

표쪽조종체에 통보문을 보내기

SendMessage( )함수를 사용하여 표쪽조종체에 통보문을 보낼수 있다. 표쪽조종체에 보내는 주요한 통보문들을 표 12-2 에 보여 주었다. SendMessage( )의 사용을 대신하여 표쪽조종체에 통보문을 보내는 처리를 간략화해 주는 몇개의 마크로가 제공되고 있다.

표 12-2. 표쪽조종체의 주요한 통보문

통 보 문	의 미
TCM_ADJUSTRECT	표쪽조종체의 표시령역과 창문의 크기를 엇바꾸어 계산한다. wParam 에 조작내용을 설정한다. 령 아닌 값을 설정하면 보내여 진 표쪽조종체의 표시령역의 크기로부터 창문의 크기가 계산된다. 령을 설정하면 보내여 진 창문의 크기로부터 표쪽조종체의 표시령역의 크기가 계산된다. lParam 에는 령역의 크기를 가리키는 RECT 구조체의 지시자를 설정한다.(이것은 창문의 크기 혹은 표쪽조종체의 표시령역의 크기로 된다.) 이 구조체에 계산결과도 돌려 진다.
TCM_DELETEALLITEMS	표쪽조종체의 모든 표쪽을 삭제한다. 호출이 성공하면 령 아닌 값을 돌려 주며 실패하면 령을 돌려 준다. wParam 과 lParam 에는 령을 설정한다.
TCM_DELETEITEM	지정한 표쪽을 삭제한다. 호출이 성공하면 령 아닌 값이 돌려 지고 실패하면 령이 돌려 진다. wParam 에는 삭제할 표쪽의 색인을 설정한다. lParam 에는 령을 설정한다.

TCM_GETCURSEL	현재 선택되여 있는 표쪽의 색인이 돌려 진다. 표쪽이 선택되여 있지 않는 경우에는 -1 이 돌려 진다. wParam 과 lParam 에는 령을 설정한다.
TCM_GETITEM	특정한 표쪽의 정보를 얻는다. 호출이 성공하면 령 아닌 값이 돌려 지고 실패하면 령이 돌려 진다. wParam 에 표쪽의 색인을 설정한다. lParam 에는 표쪽의 정보를 보관하기 위한 TCITEM 구조체의 지시자를 설정한다.
TCM_GETITEMCOUNT	표쪽의 수를 돌려 준다. wParam 과 lParam 에는 령을 설정한다.
TCM_INSERTITEM	새로운 표쪽을 작성(삽입)한다. 호출이 성공하면 삽입된 표쪽의 색인이 돌려 지고 실패하면 -1 이 돌려 진다. wParam 에는 삽입할 표쪽의 색인을 설정한다. lParam 에는 표쪽의 정보를 보관한 TCITEM 구조체의 지시자를 설정한다.
TCM_SETCURSEL	표쪽을 선택한다. 마지막으로 선택되여 있던 표쪽의 색인이 돌려 진다. 마지막으로 선택되여 있던 표쪽이 없는 경우에는 -1 이 돌려 진다. 새로 선택하는 표쪽의 색인을 설정한다. lParam 에는 령을 설정한다.
TCM_SETITEM	특정한 표쪽의 정보를 설정한다. 호출이 성공하면 령 아닌 값을 돌려 주고 실패하면 령이 돌려 진다. wParam 에는 표쪽의 색인을 설정한다. lParam 에는 표쪽의 정보를 보관한 TCITEM 구조체의 지시자를 설정한다.

표 12-2 에 보여 준 통보문을 보내는 마크로들은 다음과 같다. 모든 마크로들에서 hTabWnd 에는 표쪽조종체의 손잡이를 설정한다.

```
VOID TabCtrl_AdjustRect(HWND hTabWnd, BOOL operation,
                              RECT *lpRect);
BOOL TabCtrl_DeleteAllItems(HWND hTabwnd);
BOOL TabCtrl_DeleteItem(HWND hTabwnd, int index);
int TabCtrl_GetCurSel(HWND hTabWnd);
BOOL TabCtrl_GetItem(HWND hTabWnd, int index, TCITEM *lpitem);
int TabCtrl_ GetItemCount(HWND hTabWnd);
int TabCtrl_ InsertItem(HWND hTabwnd, int index,
```

```
                              CONST TCITEM *lpitem);
int TabCtrl_ SetCurSel(HWND hTabWnd, int index);
BOOL TabCtrl_SetItem(HWND hTabWnd, int index,
                              TCITEM *lpitem);
```

일반적으로 SendMessage( )를 사용하는것보다 마크로를 사용하는 편이 간단하다.

표쪽조종체를 작성한 시점에서는 표쪽이 없다. 그러므로 적어도 한번이상 *TCM_INSERTITEM* 통보문을 보내야 한다.

이 장의 실례프로그람에서는 사용되지 않지만 실제의 응용프로그람에서는 *TCM_ADJUSTRECT* 통보문을 사용하여 표쪽조종체의 표시령역의 크기를 얻을수 있다.

표쪽조종체를 작성할 때는 창문에 표쪽자체만이 아니라 어떤 정보나 대화칸을 표시하기 위한 령역도 필요하게 된다. 이 표시령역은 표쪽조종체의 창문에서 표쪽을 제외한 부분이다.(다시말하여 어떤 정보를 표시하기 위하여 표쪽조종체의 표시령역을 사용할수 있다.) 표쪽에 대응한 정보를 표시하는 표시령역의 크기를 얻어야 하는 경우도 있는것이다.

표쪽조종체의 통지문

사용자가 표쪽조종체를 조작하면 *WM_NOTIFY* 통보문이 생성된다. 표쪽조종체는 선택상태의 변화를 두개의 통지문으로 알려 준다. 그것들은 *TCN_SELCHANGE* 과 *TCN_SELCHANGING* 통보문이다.

TCN_SELCHANGING 은 표쪽의 선택상태가 변경되기 직전에 발송된다. TCN_SELCHANGE 는 새로운 표쪽이 선택된후에 발송된다.

WM_NOTIFY 통보문이 보내 졌을 때는 lParam 에 *NMHDR 구조체*(제 10 장에서 설명)의 지시자가 보관된다. 통지문은 NMHDR 구조체의 code 성원에 보관된다. 통보문을 생성한 표쪽조종체의 손잡이는 hwndFrom 성원에 보관된다.

표쪽조종체의 간단한 실례프로그람

표쪽조종체의 사용방법을 보여 주는 간단한 프로그람을 실례 12-2 에 보여 주었다. 이 프로그람에서는 [Options], [View] 및 [Errors]라는 세개의 표쪽을 작성하고 있다. 새로운 표쪽이 선택되면 그것을 알리는 통보가 표시된다. 프로그람의 실행결과를 그림 12-2 에 보여 주었다.

실례 12-2. Tab 프로그람

```
// 표쪽조종체의 간단한 실례프로그람

#include <windows.h>
#include <commctrl.h>
#include <cstdio>
```

```
LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);

char szWinName[] = "MyWin"; // 창문클라스의 이름

HINSTANCE hInst;
HWND hwnd;
HWND hTabWnd;

char TabName[][40] = {
  "Options",
  "View",
  "Errors"
};

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  MSG msg;
  WNDCLASSEX wcl;
  INITCOMMONCONTROLSEX cc;

  // 창문클라스를 정의한다.
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst;      // 실체의 손잡이
  wcl.lpszClassName = szWinName; // 창문클라스의 이름
  wcl.lpfnWndProc = WindowFunc;  // 창문함수
  wcl.style = 0;                 // 체계설정의 형식

  wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
  wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
  wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

  wcl.lpszMenuName = NULL; // 클라스차림표는 없다.
  wcl.cbClsExtra = 0; // 보조기억기령역은 필요 없다.
  wcl.cbWndExtra = 0;

  // 창문의 배경색을 흰색으로 한다.
  wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

  // 창문클라스를 등록한다.
  if(!RegisterClassEx(&wcl)) return 0;
```

```
  /* 창문클라스가 등록되였으므로
     창문을 작성할수 있다. */
  hwnd = CreateWindow(
    szWinName, // 창문클라스의 이름
    "Using a Tab Control", // 제목
    WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
    CW_USEDEFAULT, // X 자리표는 Windows 가 결정하게 한다.
    CW_USEDEFAULT, // Y 자리표는 Windows 가 결정하게 한다.
    CW_USEDEFAULT, // 너비는 Windows 가 결정하게 한다.
    CW_USEDEFAULT, // 높이는 Windows 가 결정하게 한다.
    NULL,          // 어미창문은 없다.
    NULL,          // 차림표는 없다.
    hThisInst,     // 실체의 손잡이
    NULL           // 추가파라메터는 없다.
  );

  // 공통조종체를 초기화한다.
  cc.dwSize = sizeof(INITCOMMONCONTROLSEX);
  cc.dwICC = ICC_TAB_CLASSES;
  InitCommonControlsEx(&cc);

  // 창문을 표시한다.
  ShowWindow(hwnd, nWinMode);
  UpdateWindow(hwnd);

  // 통보문순환고리를 작성한다.
  while(GetMessage(&msg, NULL, 0, 0))
  {
    TranslateMessage(&msg); // 건반통보를 변환한다.
    DispatchMessage(&msg); // Windows 2000 에 조종을 넘긴다.
  }

  return msg.wParam;
}

/* 이 함수는 Windows 2000 으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받아 들인다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
```

```
{
  NMHDR *nmptr;
  int tabnumber;
  HDC hdc;
  char str[80];
  RECT WinDim;
  TCITEM tci;

  switch(message) {
    case WM_CREATE:
      // 어미창문의 크기를 얻는다.
      GetClientRect(hwnd, &WinDim);

      // 표쪽조종체를 작성한다.
      hTabWnd = CreateWindow(
                  WC_TABCONTROL,
                  "",
                  WS_VISIBLE | WS_TABSTOP | WS_CHILD,
                  0, 0, WinDim.right, WinDim.bottom,
                  hwnd,
                  NULL,
                  hInst,
                  NULL
                );

      tci.mask = TCIF_TEXT;
      tci.iImage = -1;

      tci.pszText = TabName[0];
      TabCtrl_InsertItem(hTabWnd, 0, &tci);

      tci.pszText = TabName[1];
      TabCtrl_InsertItem(hTabWnd, 1, &tci);

      tci.pszText = TabName[2];
      TabCtrl_InsertItem(hTabWnd, 2, &tci);
      break;
    case WM_NOTIFY: // 표쪽의 변경을 처리한다.
      nmptr = (LPNMHDR) lParam;
      if(nmptr->code == TCN_SELCHANGE) {
        tabnumber = TabCtrl_GetCurSel((HWND)nmptr->hwndFrom);
```

```
        hdc = GetDC(hTabWnd);
        sprintf(str, "Changed to %s Tab.      ",
                TabName[tabnumber]);
        SetBkColor(hdc, RGB(200, 200, 200));
        TextOut(hdc, 40, 100, str, strlen(str));
        ReleaseDC(hTabWnd, hdc);
      }
      break;
    case WM_DESTROY: // 프로그람을 끝낸다.
      PostQuitMessage(0);
      break;
    default:
      /* 이 switch 문에서 지정된것 이외의 통보문은
         Windows 2000 에 처리를 맡긴다. */
     return DefWindowProc(hwnd, message, wParam, lParam);
  }

  return 0;
}
```

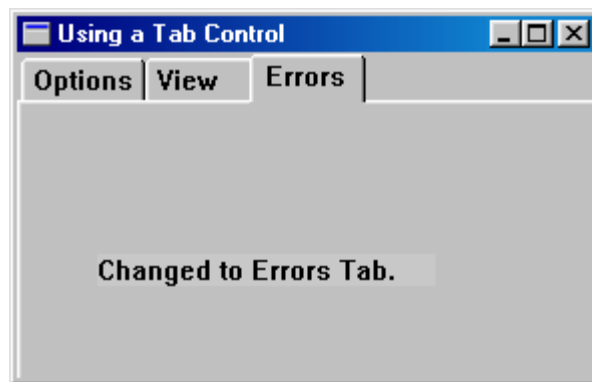



그림 12-2. 표쪽조종체의 첫 실례프로그람의 실행결과

이 프로그람에서는 표쪽조종체가 작성되기전에 GetClientRect()를 사용하여 기본창문의 크기를 얻고 있다. 표쪽조종체를 작성할 때 그 크기가 어미창문의 의뢰자구역전체를 차지하게 하고 있다. 표쪽조종체의 크기는 임의로 해도 상관 없지만 이렇게 하는 경우도 흔히 있다. 표쪽조종체를 작성하고 나서 거기에 세개의 표쪽을 삽입하고 있다.

TCN_SELCHANGE 를 포함한 WM_NOTIFY 통보문을 받으면 표쪽조종체의 표시령역에 표쪽의 선택상태가 변경되였다는것을 알리는 통보를 표시한다. 새로 선택한 표쪽의 색인은 TCM_GETCURSEL 통보문을 보내여 얻을수 있다.

표쪽조종체의 사용방법

표쪽조종체를 작성하는것은 간단하지만 매개 표쪽에 대화칸을 련관시키는 약간 기교적인 사용방법도 적용할수 있다. 새로운 표쪽이 선택되면 그때까지 표시되여 있던 대화칸이 없어지고 새로운 대화칸이 표시된다.

매 대화칸을 표쪽조종체의 표시령역의 크기에 맞추고 싶다고 생각할것이다. 이런 문제를 비롯하여 표쪽조종체와 대화칸을 련관시키는 처리에는 복잡한 문제점들이 많이 존재하며 그것들을 이 책에서 모두 해결하는것은 불가능하다.

여기에서는 표쪽조종체의 기본적인 사용방법만을 설명한다. 본질을 파악하면 필요한때 자체로 표쪽조종체에 여러가지 기능을 추가할수 있을것이다.

아무때나 새로운 표쪽을 선택할수 있게 하려면 *비양식화대화칸*을 사용하여야 한다. 그것은 비양식화대화칸이 대화칸을 표시한 상태에서도 응용프로그람의 다른 부분을 능동으로 할수 있기때문이다.(이것은 프로그람의 다른 부분을 사용하려면 대화칸을 닫아야하는 *양식화대화칸*과 다른 점이다.)

새로운 표쪽이 선택되였을 때는 마지막으로 표시되여 있던 대화칸을 소거하고 새로운 대화칸을 능동으로 한다. 아무때나 대화칸을 닫겨 지게 하려면 새로운 대화칸을 표시하기전에 응용프로그람에서 적절한 처리(실례로 현재의 설정상태를 보관하는것 등)를 진행하여야 한다.

실례 12-3 의 프로그람은 앞에서 작성한 실례프로그람을 개조하여 세개의 표쪽에 대화칸을 표시하는것이다. 매 표쪽에는 자체의 비양식화대화칸이 련관되여 있다. 첫 대화칸은 이 장의 앞부분에서 작성한 상태창문의 실례프로그람에서 사용된 대화칸이다. 다른 두 대화칸은 그 어떤 기능도 수행하지 않는 허위적인것으로 작성되였다. 프로그람의 실행결과를 그림 12-3 에 보여 주었다.

실례 12-3. Tab2 프로그람

```
// 표쪽조종체의 실례

#include <windows.h>
#include <commctrl.h>
#include <cstdio>
#include "tab.h"

#define NUMPARTS 5

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK DialogFunc1(HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK DialogFunc2(HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK DialogFunc3(HWND, UINT, WPARAM, LPARAM);
```

```
void InitStatus(HWND hwnd);

char szWinName[] = "MyWin"; // 창문클라스의 이름

HINSTANCE hInst;
HWND hwnd;
HWND hStatusWnd;
HWND hTabWnd;

int parts[NUMPARTS];

HWND hDlg = (HWND) NULL;

char TabName[][40] = {
  "Options",
  "View",
  "Errors"
};

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  MSG msg;
  WNDCLASSEX wcl;
  INITCOMMONCONTROLSEX cc;

  // 창문클라스를 정의한다.
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst;      // 실체의 손잡이
  wcl.lpszClassName = szWinName; // 창문클라스의 이름
  wcl.lpfnWndProc = WindowFunc;  // 창문함수
  wcl.style = 0;                 // 체계설정의 형식

  wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
  wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
  wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식
```

```
wcl.lpszMenuName = NULL; // 클라스차림표는 없다.
wcl.cbClsExtra = 0; // 보조기억기령역은 필요 없다.
wcl.cbWndExtra = 0;

// 창문의 배경색을 흰색으로 한다.
wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

// 창문클라스를 등록한다.
if(!RegisterClassEx(&wcl)) return 0;

/* 창문클라스가 등록되였으므로
   창문을 작성할수 있다. */
hwnd = CreateWindow(
  szWinName, // 창문클라스의 이름
  "Using a Tab Control", // 제목
  WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
  CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 너비는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 높이는 Windows가 결정하게 한다.
  NULL,          // 어미창문은 없다.
  NULL,          // 차림표는 없다.
  hThisInst,     // 실체의 손잡이
  NULL           // 추가파라메터는 없다.
);

// 공통조종체를 초기화한다.
cc.dwSize = sizeof(INITCOMMONCONTROLSEX);
cc.dwICC = ICC_TAB_CLASSES | ICC_UPDOWN_CLASS;
InitCommonControlsEx(&cc);

hInst = hThisInst; 현재실체손잡이를 보관한다.

// 창문을 표시한다.
ShowWindow(hwnd, nWinMode);
UpdateWindow(hwnd);

// 통보문순환고리를 작성한다.
```

```
  while(GetMessage(&msg, NULL, 0, 0))
  {
    if(!hDlg || !IsDialogMessage(hDlg, &msg)) {
      TranslateMessage(&msg); // 건반통보를 변환한다.
      DispatchMessage(&msg); // Windows 2000 에 조종을 넘긴다.
    }
  }

  return msg.wParam;
}

/* 이 함수는 Windows 2000 으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받아 들인다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  NMHDR *nmptr;
  int tabnumber = 0;
  TCITEM tci;

  switch(message) {
    case WM_CREATE:
      hTabWnd = CreateWindow(
                  WC_TABCONTROL,
                  "",
                  WS_VISIBLE | WS_TABSTOP | WS_CHILD,
                  20, 20, 368, 246,
                  hwnd,
                  NULL,
                  hInst,
                  NULL
                );

      tci.mask = TCIF_TEXT;
      tci.iImage = -1;

      tci.pszText = TabName[0];
```

```
    TabCtrl_InsertItem(hTabWnd, 0, &tci);

    tci.pszText = TabName[1];
    TabCtrl_InsertItem(hTabWnd, 1, &tci);

    tci.pszText = TabName[2];
    TabCtrl_InsertItem(hTabWnd, 2, &tci);

    hDlg = CreateDialog(hInst, "TabDB1", hTabWnd,
                        (DLGPROC) DialogFunc1);
    break;
  case WM_NOTIFY:
    nmptr = (LPNMHDR) lParam;
    if(nmptr->code == TCN_SELCHANGE) {
      if(hDlg) DestroyWindow(hDlg);
      tabnumber = TabCtrl_GetCurSel((HWND)nmptr->hwndFrom);
      switch(tabnumber) {
        case 0:
          hDlg = CreateDialog(hInst, "TabDB1",
                     hTabWnd, (DLGPROC) DialogFunc1);
          break;
        case 1:
          hDlg = CreateDialog(hInst, "TabDB2",
                     hTabWnd, (DLGPROC) DialogFunc2);
          break;
        case 2:
          hDlg = CreateDialog(hInst, "TabDB3",
                     hTabWnd, (DLGPROC) DialogFunc3);
          break;
      }
    }
    break;
  case WM_DESTROY: // 프로그람을 끝낸다.
    if(hDlg) DestroyWindow(hDlg);
    PostQuitMessage(0);
    break;
  default:
    /* 이 switch 문에서 지정된것 이외의 통보문은
```

```
          Windows 2000에 처리를 맡긴다. */
      return DefWindowProc(hwnd, message, wParam, lParam);
    }
    return 0;
  }

  // 「Options」의 대화함수
  BOOL CALLBACK DialogFunc1(HWND hdwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
  {
    static long udpos = 0;
    static char str[80];
    static HWND hEboxWnd;
    static HWND udWnd;
    static statusCB1, statusCB2;
    static statusRB1;
    int low=0, high=10;

    switch(message) {
      case WM_INITDIALOG:
        InitStatus(hdwnd); // 상태창문을 초기화한다.

        hEboxWnd = GetDlgItem(hdwnd, ID_EB1);
        udWnd = CreateUpDownControl(
                      WS_CHILD | WS_BORDER | WS_VISIBLE |
                      UDS_SETBUDDYINT | UDS_ALIGNRIGHT,
                      10, 10, 50, 50,
                      hdwnd,
                      ID_UPDOWN,
                      hInst,
                      hEboxWnd,
                      high, low, high/2);

        // 단일선택단추를 초기화한다.
        SendDlgItemMessage(hdwnd, ID_RB2, BM_SETCHECK,
                           BST_CHECKED, 0);
        return 1;
      case WM_VSCROLL: // 오르내리기조종체를 초기화한다.
```

```
      if(udWnd==(HWND)lParam) {
        udpos = GetDlgItemInt(hdwnd, ID_EB1, NULL, 1);
        sprintf(str, "Power: %d", udpos);
        SendMessage(hStatusWnd, SB_SETTEXT,
                    (WPARAM) 0, (LPARAM) str);
      }
      return 1;
    case WM_COMMAND:
      switch(LOWORD(wParam)) {
        case ID_CB1: // 「Optimize」검사칸을 처리한다.
          statusCB1 = SendDlgItemMessage(hdwnd, ID_CB1,
                      BM_GETCHECK, 0, 0);
          if(statusCB1) sprintf(str, "Optimize: Yes");
          else sprintf(str, "Optimize: No");
          SendMessage(hStatusWnd, SB_SETTEXT,
                    (WPARAM) 1, (LPARAM) str);
          return 1;
        case ID_CB2: // 「Debug」검사칸을 처리한다.
          statusCB2 = SendDlgItemMessage(hdwnd, ID_CB2,
                      BM_GETCHECK, 0, 0);
          if(statusCB2) sprintf(str, "Debug: Yes");
          else sprintf(str, "Debug: No");
          SendMessage(hStatusWnd, SB_SETTEXT,
                    (WPARAM) 2, (LPARAM) str);
          return 1;
        case ID_CB3: // 「NT 4」검사칸을 처리한다.
          statusCB2 = SendDlgItemMessage(hdwnd, ID_CB3,
                      BM_GETCHECK, 0, 0);
          if(statusCB2) sprintf(str, "NT 4: Yes");
          else sprintf(str, "NT 4: No");
          SendMessage(hStatusWnd, SB_SETTEXT,
                    (WPARAM) 3, (LPARAM) str);
          return 1;
        case ID_RB1: // 단일선택단추를 처리한다.
        case ID_RB2:
          statusRB1 = SendDlgItemMessage(hdwnd, ID_RB1,
                      BM_GETCHECK, 0, 0);
          if(statusRB1) sprintf(str, "Using C");
```

```
          else sprintf(str, "Using C++");
          SendMessage(hStatusWnd, SB_SETTEXT,
                      (WPARAM) 4, (LPARAM) str);
          return 1;
        case ID_RESET: // 추가선택항목을 재설정한다.
          SendMessage(udWnd, UDM_SETPOS, 0, (LPARAM) high /2);
          SendMessage(hStatusWnd, SB_SETTEXT, (WPARAM) 0,
              (LPARAM) "Power: 5");
          SendDlgItemMessage(hdwnd, ID_CB1,
                      BM_SETCHECK, 0, 0);
          SendDlgItemMessage(hdwnd, ID_CB2,
                      BM_SETCHECK, 0, 0);
          SendDlgItemMessage(hdwnd, ID_CB3,
                      BM_SETCHECK, 0, 0);
          SendMessage(hStatusWnd, SB_SETTEXT, (WPARAM) 1,
              (LPARAM) "Optimize: No");
          SendMessage(hStatusWnd, SB_SETTEXT, (WPARAM) 2,
              (LPARAM) "Debug: No");
          SendMessage(hStatusWnd, SB_SETTEXT, (WPARAM) 3,
              (LPARAM) "NT 4: No");
          SendMessage(hStatusWnd, SB_SETTEXT, (WPARAM) 4,
              (LPARAM) "Using C++");
          return 1;
        case IDCANCEL:
        case IDOK:
          PostQuitMessage(0);
          return 1;
      }
  }
  return 0;
}

// 두번째 대화함수(허위적인것)
BOOL CALLBACK DialogFunc2(HWND hdwnd, UINT message,
                          WPARAM wParam, LPARAM lParam)
{
  switch(message) {
    case WM_COMMAND:
```

```
      switch(LOWORD(wParam)) {
        case IDCANCEL:
        case IDOK:
          PostQuitMessage(0);
          return 1;
      }
  }
  return 0;
}

// 세번째 대화함수(허위적인것)
BOOL CALLBACK DialogFunc3(HWND hdwnd, UINT message,
                          WPARAM wParam, LPARAM lParam)
{
  switch(message) {
    case WM_COMMAND:
      switch(LOWORD(wParam)) {
        case IDCANCEL:
        case IDOK:
          PostQuitMessage(0);
          return 1;
      }
  }
  return 0;
}

// 상태창문의 초기화
void InitStatus(HWND hwnd)
{
  RECT WinDim;
  int i;

  GetClientRect(hwnd, &WinDim);

  for(i=1; i<=NUMPARTS; i++)
    parts[i-1] = WinDim.right/NUMPARTS * i;

  // 상태창문을 작성한다.
```

```
    hStatusWnd = CreateStatusWindow(
                WS_CHILD | WS_VISIBLE,
                "Power: 5", hwnd,
                ID_STATUSWIN);

    SendMessage(hStatusWnd, SB_SETPARTS,
              (WPARAM) NUMPARTS, (LPARAM) parts);

    SendMessage(hStatusWnd, SB_SETTEXT, (WPARAM) 1,
              (LPARAM) "Optimize: No");
    SendMessage(hStatusWnd, SB_SETTEXT, (WPARAM) 2,
              (LPARAM) "Debug: No");
    SendMessage(hStatusWnd, SB_SETTEXT, (WPARAM) 3,
              (LPARAM) "NT 4: No");
    SendMessage(hStatusWnd, SB_SETTEXT, (WPARAM) 4,
              (LPARAM) "Using C++");
  }
```

이 프로그람에는 아래의 자원이 필요하다. 경계선이 없는 대화칸이라는 점에 주의를 돌려야 한다.

```
  #include <windows.h>
  #include "tab.h"

  TabDB1 DIALOGEX 2, 16, 180, 92
  STYLE  WS_CHILD | WS_VISIBLE
  {
    PUSHBUTTON "Reset", ID_RESET, 112, 40, 37, 14
    PUSHBUTTON "OK", IDOK, 112, 60, 37, 14
    EDITTEXT ID_EB1, 10, 10, 20, 12, ES_LEFT |
             WS_TABSTOP | WS_BORDER
    LTEXT "Power Factor", ID_STEXT1, 36, 12, 50, 12
    AUTOCHECKBOX "Optimize", ID_CB1, 10, 35, 48, 12
    AUTOCHECKBOX "Debug Info", ID_CB2, 10, 50, 48, 12
    AUTOCHECKBOX "NT 4 Compatible", ID_CB3, 10, 65, 70, 12
    AUTORADIOBUTTON "C", ID_RB1, 112, 8, 20, 12
    AUTORADIOBUTTON "C++", ID_RB2, 112, 20, 28, 12
    LTEXT "Language", ID_STEXT2, 140, 14, 40, 12
```

```
}

TabDB2 DIALOGEX 2, 16, 180, 92
STYLE WS_CHILD | WS_VISIBLE
{
  PUSHBUTTON "OK", IDOK, 82, 43, 37, 14
  PUSHBUTTON "CANCEL", IDCANCEL, 132, 43, 37, 14
  LTEXT "Show contents of", ID_STEXT3, 10, 10, 60, 12
  AUTORADIOBUTTON "Registers", ID_RB3, 10, 30, 48, 12
  AUTORADIOBUTTON "Variables", ID_RB4, 10, 50, 52, 12
  AUTORADIOBUTTON "Stack", ID_RB5, 10, 70, 48, 12
}

TabDB3 DIALOGEX  2, 16, 180, 92
STYLE WS_CHILD | WS_VISIBLE
{
  PUSHBUTTON "OK", IDOK, 82, 43, 37, 14
  PUSHBUTTON "CANCEL", IDCANCEL, 132, 43, 37, 14
  LTEXT "Report", ID_STEXT4, 10, 10, 60, 12
  AUTOCHECKBOX "Syntax errors", ID_CB4, 10, 30, 60, 12
  AUTOCHECKBOX "Warnings", ID_CB5, 10, 50, 60, 12
  AUTOCHECKBOX "ANSI Violations", ID_CB6, 10, 70, 70, 12
}
```

머리부파일 TAB.H 의 내용을 아래에 보여 주었다.

```
#define IDM_DIALOG          100
#define IDM_EXIT            101
#define IDM_HELP            102
#define ID_UPDOWN           103
#define ID_EB1              104
#define ID_EB2              105
#define ID_CB1              106
#define ID_CB2              107
#define ID_CB3              108
#define ID_CB4              109
#define ID_CB5              110
#define ID_CB6              111
```

```
#define ID_RB1          112
#define ID_RB2          113
#define ID_RB3          114
#define ID_RB4          115
#define ID_RB5          116
#define ID_RESET        117

#define ID_STATUSWIN    200

#define ID_STEXT1       300
#define ID_STEXT2       301
#define ID_STEXT3       302
#define ID_STEXT4       303
```

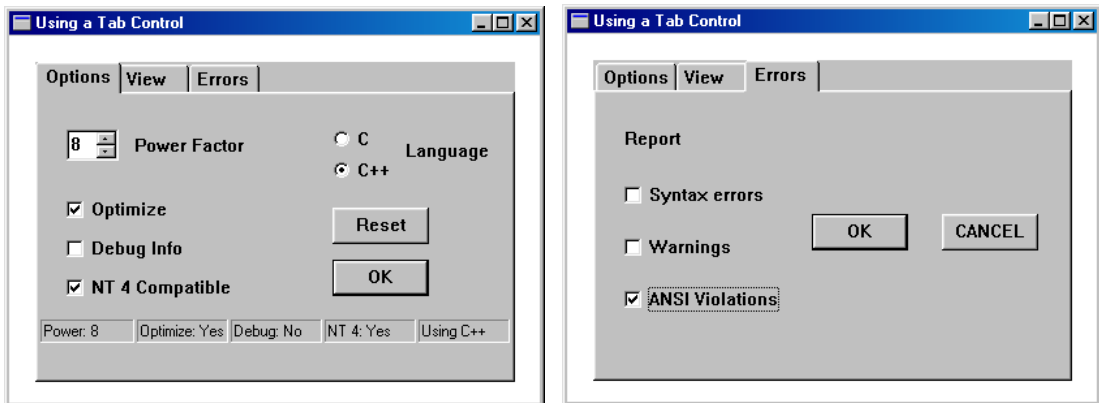



그림 12-3. 표쪽조종체의 두번째 실례프로그람의 실행결과

이 프로그람에서 특히 주목해야 할 부분은 WindowsFunc( )에서 WM_NOTIFY 통보문을 처리하는 case 문이다. 아래에 다시 그 부분의 프로그람코드를 보여 주었다.

```
    case WM_NOTIFY:
        nmptr = (LPNMHDR) lParam;
        if(nmptr->code == TCN_SELCHANGE) {
          if(hDlg) DestroyWindow(hDlg);
          tabnumber = TabCtrl_GetCurSel((HWND)nmptr->hwndFrom);
          switch(tabnumber) {
            case 0:
              hDlg = CreateDialog(hInst, "TabDB1",
                                  hTabWnd, (DLGPROC) DialogFunc1);
```

```
            break;
          case 1:
            hDlg = CreateDialog(hInst, "TabDB2",
                          hTabWnd, (DLGPROC) DialogFunc2);
            break;
          case 2:
            hDlg = CreateDialog(hInst, "TabDB3",
                          hTabWnd, (DLGPROC) DialogFunc3);
            break;
        }
      }
    break;
```

새로운 표쪽이 선택되였을 때 두가지 처리가 진행된다. 우선 DestroyWindow( )를 사용하여 현재 선택되여 있는 표쪽에 련관된 대화칸을 삭제한다.(비양식화대화칸을 닫으려면 양식화대화칸에서 사용되는 EndDialog( )가 아니라 DestroyWindow( )를 사용하여야 한다.)

다음 선택된 표쪽을 얻고 그 표쪽에 련관된 대화칸을 CreateDialog( )를 사용하여 작성한다.(비양식화대화칸을 작성하는데 CreateDialog( )를 사용한다.) 이 처리순서를 자체로 작성하는 응용프로그람에서 표쪽조종체와 대화칸을 련관시킬 때도 사용할수 있다.

더 나가기에 앞서 이 프로그람을 가지고 여러가지 실험을 해보는것이 유익하다. 례하면 대화칸이나 표쪽조종체를 변경해 보시오. 표쪽조종체에 도구설명쪽지를 추가할수도 있다.

나무구조보기조종체

나무구조보기조종체(Tree View Control)는 나무구조로 정보를 표시하는데 사용된다. 례를 들어 Windows 2000 의 탐색프로그람(Internet Explorer)에서는 등록부의 목록을 나무구조보기조종체를 사용하여 표시하고 있다. 나무는 계층구조로 되여 있으므로 계층적인 정보를 표시하는 경우에만 나무구조보기조종체를 사용해야 한다. 나무구조보기조종체는 매우 강력하며 많은 추가선택기능들을 제공하고 있다. 나무구조보기조종체에 대한 설명만으로도 한권의 책이 될것이다.

그러므로 여기에서는 나무구조보기조종체의 기본적인 사용방법에 대해서만 설명하기로 한다. 본질을 알면 자체로 나무구조보기조종체에 여러가지 기능들을 추가할 수 있다.

나무구조보기조종체의 작성

나무구조보기조종체는 *WC_TREEVIEW* 클라스를 지정하고 CreateWindow( ) 또는 CreateWindowEx( )를 사용하여 작성하는 창문이다. 나무구조보기조종체는 새끼창문이므로 *WS_CHILD* 형식도 지정해야 한다.

일반적으로 나무구조보기조종체가 자동적으로 표시되게 하기 위해 WS_VISIBLE 형식도 지정한다. *WS_TABSTOP* 형식도 지정해야 한다. 나무구조보기조종체에는 나무와 관련된 형식을 몇가지 지정할수 있다. 아래에 형식의 종류를 보여 주었다.

형　　식	의　　미
TVS_HASLINES	나무가지에 선을 표시한다.
TVS_LINESATROOT	뿌리와 가지를 련결하는 선을 표시한다.
TVS_HASBUTTONS	매 가지의 왼쪽에 전개/접기단추를 표시한다.

TVS_HASLINES 형식과 *TVS_LINESATROOT* 형식을 포함시켜 나무의 매 항목에로 뻗는 직선을 표시할수 있다. 이렇게 하면 나무구조보기조종체가 실지 나무 같은 모양을 가지게 된다.

TVS_HASBUTTONS 를 포함시키면 표준적인 전개/접기단추가 추가된다. 가지를 한 단계이상 전개할수 있는 경우에는 단추에 +표식이 표시된다. 이 단추를 마우스로 누르면 가지를 전개하거나 접을수 있다.

나무구조보기조종체를 작성할 때는 이 세개의 형식을 모두 포함시키는것이 일반적이다. 표준적인 나무구조보기조종체를 작성하기 위한 프로그람코드의 례를 아래에 보여 주었다.

```
hTreeWndCtrl = CreateWindow(
                    WC_TREEVIEW,
                    "",
                    WS_VISIBLE | WS_TABSTOP | WS_CHILD |
                    TVS_HASLINES | TVS_HASBUTTONS |
                    TVS_LINESATROOT,
                    0, 0, 100, 100,
                    hwnd, // 어미손잡이
                    NULL,
                    hInst,   // 실체손잡이
                    NULL
          );
```

나무구조보기조종체를 작성한 직후에는 그 내용이 비여 있다. 나무에 항목을 추가하는 방법은 다음에 설명한다.

나무구조보기조종체에 통보문을 보내기

나무구조보기조종체에 몇가지 통보문을 보낼수 있다. 나무구조보기조종체의 주요한 통보문들을 표 12-3 에 보여 주었다. 통보문을 보내기 위해 SendMessage( )를 사용하든가 전용마크로를 사용한다. 다음에 보여 준 나무구조보기조종체용마크로들은 표 12-3 에 보여 준 통보문들을 보내는 기능을 수행한다. 이 모든 마크로들에 있어서 hTreeWnd 에는 나무구조보기조종체의 손잡이를 설정한다.

```
BOOL TreeView_DeleteItem(, HTREEITEM hItem)

BOOL TreeView_Expand(HWND hTreeWnd, HTREEITEM hItem,
                                        UINT action)

BOOL TreeView_GetItem(TVITEM *lpItem)

HTREEVIEW TreeView_InsertItem(HWND hTreeWnd,
                                        TVINSERTSTRUCT * lpItem)

BOOL TreeView_Select(HWND hTreeWnd, HTREEITEM hItem,
                                        UINT action)
```

이 장의 실례프로그람에서 사용되는 통보문들은 TVM_INSERTITEM 과 TVM_EXPAND 이다.

표 12-3. 나무구조보기조종체의 주요한 통보문

통 보 문	의 미
TVM_DELETEITEM	나무목록에서 항목을 삭제한다. 호출이 성공하면 령 아닌 값을 돌려 주고 실패하면 령을 돌려 준다. wParam 에는 령을 설정한다. lParam 에는 삭제할 항목의 손잡이를 설정한다.

TVM_EXPAND	나무목록을 한 단계 전개하거나 접는다. 호출이 성공하면 령 아닌 값을 돌려 주고 실패하면 령을 돌려 준다. wParam 에는 조작내용을 설정한다. 이 값은 TVE_COLLAPSE(나무를 접기), TVE_COLLAPSE RESET(나무를 접고 새끼항목을 삭제한다.), TVE_EXPAND(나무를 전개한다.), TVE_EXPAND PARTIAL(나무를 부분적으로 전개한다.) 또는 TVE_TOGGLE(상태를 반전절환하여 변경한다.)의 어느 하나여야 한다.TVE_COLLAPERESET 는 TVE_COLLAPSE 와 함께 사용되여야 한다. TVE_EXPANDPARTIAL 은 TVE_EXPAND 와 함께 사용되여야 한다. lParam 에는 가지어미의 손잡이를 설정한다.
TVM_GETITEM	항목의 속성을 얻는다. 호출이 성공하면 령 아닌 값을 돌려 주고 실패하면 령을 돌려 준다. wParam 에는 령을 설정한다. lParam 에는 항목의 정보를 보관하기 위한 TVITEM 구조체의 지시자를 설정한다.
TVM_INSERTITEM	나무에 항목을 삽입한다. 호출이 성공하면 삽입된 항목의 손잡이를 돌려 주고 실패하면 NULL 을 돌려 준다. wParam 에는 령을 설정한다. lParam 에는 항목의 정보를 보관하기 위한 TVINSERTSTRUCT 구조체의 지시자를 설정한다.
TVM_SELECTITEM	나무구조보기의 항목을 선택한다. 호출이 성공하면 령 아닌 값이 돌려 주고 실패하면 령을 돌려 준다. wParam 에는 조작내용을 설정한다. 이 값이 TVGN_CARET 인 경우는 항목이 선택된다. TVGN_DROPHILITE 인 경우는 끌어다놓기(Drag and drop)를 위해 항목이 강조표시된다. TVGN_FIRSTVISIBLE 인 경우는 지정된 항목이 제일 우에 표시되게 나무구조보기가 흘리기된다. lParam 에는 항목의 손잡이를 설정한다.

항목을 삽입할 때는 아래에 보여 준 *TVINSERTSTRUCT* 구조체에 항목의 정보를 보관한다.

```
typedef struct tagTVINSERTSTRUCT {
    HTREEITEM hParent;
    HTREEITEM hInsertAfter;
```

```
        union {
                TVITEMEX item;
                TVITEM item;
        }
    }   TVINSERTSTRUCT;
```

hParent 는 항목에 있는 어미의 손잡이이다. 항목에 어미가 없는 경우는 여기에 *TVI_ROOT* 를 설정한다. hInsertAfter 의 값은 새로운 항목을 나무에 삽입하는 방법을 지정한다. 여기에 항목의 손잡이를 설정하면 그 항목의 밑에 새로운 항목이 삽입된다. 그렇지 않은 경우에는 hInsertAfter 에 아래의 어느 한 값을 설정한다.

hInsertAfter 의 값	의 미
TVI_FIRST	목록의 선두에 삽입한다.
TVI_LAST	목록의 끝에 삽입한다.
TVI_ROOT	뿌리에 삽입한다.
TVI_SORT	자모순으로 삽입한다.

item 에 항목의 내용을 설정한다. 이 성원은 TVITEM 또는 TVITEMEX 구조체여야 한다. *TVITEMEX* 는 TVITEM 의 확장판이고 이 장의 실례프로그람에서는 사용되지 않는 몇가지 특수한 기능을 제공한다. *TVITEM* 구조체의 정의를 아래에 보여 주었다.

```
    typedef struct tagTVITEM {
        UINT mask;
        HTREEITEM hItem;
        UINT state;
        UINT stateMask;
        LPSTR pszText;
        int cchTextMax;
        int iImage;
        int iSelectedImage;
        int cChildren;
        LPARAM lParam;
    }   TVITEM;
```

mask 의 값은 TVITEM 구조체에 나무구조보기조종체의 정보가 보관되였을 때 어느 성원에 유효한 값이 보관되여 있는가를 가리킨다. mask 의 값은 아래의 값들의 조합으로 된다.

mask 의 값	의　　미
TVIF_HANDLE	hItem 에 자료가 보관되여 있다.
TVIF_STATE	state 와 stateMask 에 자료가 보관되여 있다.
TVIF_TEXT	pszText 와 cchTextMax 에 자료가 보관되여 있다.
TVIF_IMAGE	iImage 에 자료가 보관되여 있다.
TVIF_SELECTEDIMAGE	iSelectedImage 에 자료가 보관되여 있다.
TVIF_CHILDREN	cChildren 에 자료가 보관되여 있다.
TVIF_PARAM	lParam 에 자료가 보관되여 있다.

hItem 에는 항목의 손잡이를 설정한다.

state 에는 나무구조보기조종체의 상태를 설정한다. 나무구조보기조종체의 주요한 상태들을 아래에 보여 주었다.

state 의 값	의　　미
TVIS_DROPHILITED	끌어다놓기의 목적지로서 항목이 강조표시되여 있다.
TVIS_EXPANDED	항목이하의 가지가 모두 전개되여 있다.(어미항목에만 적용된다.)
TVIS_EXPANDEDONCE	항목이하의 가지가 한단계(혹은 그이상) 전개되여 있다.(어미항목에만 적용된다.)
TVIS_EXPANDEDPARTIAL	항목이하의 가지가 부분적으로 전개되여 있다.
TVIS_SELECTED	항목이 선택되여 있다.

stateMask 에는 상태를 설정하거나 얻으려는 항목을 지정한다. 여기에는 앞에서 보여 준 한개이상의 값을 지정한다.

항목을 나무에 삽입할 때는 pszText 에 나무에 표시될 문자렬의 지시자를 설정한다. 항목의 정보를 얻을 때는 본문을 보관하기 위한 배렬의 지시자를 pszText 에 설정한다. 이 경우에는 cchTextMax 에 pszText 에서 가리키는 배렬의 크기를 설정한다. 이밖의 경우에 cchTextMax 는 무시된다.

나무구조보기조종체에 화상목록을 련관시키는 경우에는 항목이 선택되여 있지 않을 때 사용되는 화상의 색인을 iImage 에 설정한다. 항목이 선택되여 있을 때 사용되는 화상의 색인은 iImageSelected 에 설정한다.(이 장에서 작성하는 나무구조보기조종체에서는 화상을 사용하지 않는다.)

항목이 새끼항목을 가지는 경우에는 cChildren 에 1 을 설정한다. 그렇지 않은 경우에는 령을 설정한다. lParam 에는 응용프로그람자체의 자료를 설정할수 있다.

나무구조보기조종체의 통지문

나무구조보기조종체가 조작되면 WM_NOTIFY 통보문이 생성된다. 나무구조보기조종체와 관련된 통지문들이 몇 종류 있다. 잘 사용되는 통지문들을 아래에 보여 주었다.

통지문	의　　미
TVN_DELETEITEM	항목이 삭제되였다.
TVN_ITEMEXPANDING	가지가 전개되거나 접히려고 하고 있다.
TVN_ITEMEXPANDED	가지가 전개되거나 접히였다.
TVN_SELCHANGING	새로운 항목이 선택되려고 하고 있다.
TVN_SELCHANGED	새로운 항목이 선택되였다.

이 통지문들을 얻으려면 WM_NOTIFY 통보문을 보냈을 때 lParam 에 보관되여 있는 NMTREEVIEW 구조체의 지시자를 참조한다. NMTREEVIEW 구조체의 정의를 아래에 보여 주었다.

```
typedef struct tagNMTREEVIEW {
      NMHDR hdr;
      UINT action;
      TVITEM;
      TVITEM itemNew;
      POINT ptDrag;
}     NMTREEVIEW;
```

NMTREEVIEW 구조체의 첫 성원은 *NMHDR 구조체로* 되여 있다. 통지문은 hdr.code 에 보관된다. 그리고 통보문을 생성한 나무구조보기조종체의 손잡이는 hdr.hwndFrom 에 보관된다.

action 에는 통지문과 관련된 정보가 보관된다. itemOld 및 itemNew 에는 마지막으로 선택되여 있던 항목(만일 있다면)의 정보 또는 새로 선택된 항목(만일 있다면)의 정보가 보관된다. 통보문이 생성되였을 때의 마우스의 위치가 ptDrag 에 보관된다.

TVN_SELCHANGING 과 *TVN_SELCHANGED* 에서는 itemOld 에 마지막으로 선택되여 있던 항목이 보관되고 itemNew 에는 새롭게 선택된 항목이 보관된다. *TVN_ITEMEXPANDING* 과 *TVN_ITEMEXPANDED* 에서는 itemNew 에 전개되는 가지의 어미항목이 보관된다. *TVN_DELETEITEM* 에서는 itemOld 에 삭제된 항목이 보관된다.

이식과 관련한 요점 : 나무구조보기조종체와 관련한 구조체들의 이름은 새롭게 변경되였다. 아래의 표에 이 장에서 사용되고 있는 구조체들의 낡은 이름과 새 이름의 대응관계를 보여 주었다. 낡은 프로그람을 이식하는 경우에는 새로운 이름의 구조체를 사용해야 한다.

낡은 이름	새로운 이름
TV_ITEM	TVITEM
TV_INSERTSTRUCT	TVINSERTSTRUCT
NM_TREEVIEW	NMTREEVIEW
NM_DISPINFO	NMTVDISPINFO

나무구조보기조종체의 실례프로그람

실례 12-4 의 프로그람은 나무구조보기조종체의 실례프로그람이다. 이 프로그람에서는 간단한 나무를 작성하고 가지의 전개와 나무전체의 전개 및 가지의 접기를 할수 있다. 나무구조보기조종체에서 새로운 항목이 선택되면 그 내용이 창문에 표시된다. 프로그람의 실행결과를 그림 12-4 에 보여 주었다.

실례 12-4. Tree 프로그람

```
// 나무구조보기조종체의 실례

#include <windows.h>
#include <commctrl.h>
#include "tree.h"

#define NUM 6

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK DialogFunc(HWND, UINT, WPARAM, LPARAM);
void InitTree(void);
void report(HDC hdc, char *s);

char szWinName[] = "MyWin"; // 창문클라스의 이름

HINSTANCE hInst;
HWND hwnd;
HWND hTreeWndCtrl;
HTREEITEM hTreeWnd[NUM];
HTREEITEM hTreeCurrent;

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
```

```
MSG msg;
WNDCLASSEX wcl;
HACCEL hAccel;
INITCOMMONCONTROLSEX cc;

// 창문클라스를 정의한다.
wcl.cbSize = sizeof(WNDCLASSEX);

wcl.hInstance = hThisInst;      // 실체의 손잡이
wcl.lpszClassName = szWinName; // 창문클라스의 이름
wcl.lpfnWndProc = WindowFunc;  // 창문함수
wcl.style = 0;                 // 체계설정의 형식
wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

wcl.lpszMenuName = "TreeViewMenu"; // 기본차림표

wcl.cbClsExtra = 0; // 보조기억기령역은 필요 없다.
wcl.cbWndExtra = 0;

// 창문의 배경색을 흰색으로 한다.
wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

// 창문클라스를 등록한다.
if(!RegisterClassEx(&wcl)) return 0;

/* 창문클라스가 등록되였으므로
   창문을 작성할수 있다. */
hwnd = CreateWindow(
  szWinName, // 창문클라스의 이름
  "Using a Tree-View Control", // 제목
  WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
  CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 너비는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 높이는 Windows가 결정하게 한다.
  NULL,          // 어미창문은 없다.
  NULL,          // 차림표는 없다.
```

```
    hThisInst,     // 실체의 손잡이
    NULL            // 추가파라메터는 없다.
  );

  hInst = hThisInst; // 현재실체의 손잡이를 보관한다.

  // 건반가속기를 적재한다.
  hAccel = LoadAccelerators(hThisInst, "TreeViewMenu");

  // 공통조종체를 초기화한다.
  cc.dwSize = sizeof(INITCOMMONCONTROLSEX);
  cc.dwICC = ICC_TREEVIEW_CLASSES;
  InitCommonControlsEx(&cc);

  // 창문을 표시한다.
  ShowWindow(hwnd, nWinMode);
  UpdateWindow(hwnd);

  // 통보문순환고리를 작성한다.
  while(GetMessage(&msg, NULL, 0, 0))
  {
    if(!TranslateAccelerator(hwnd, hAccel, &msg)) {
      TranslateMessage(&msg); // 건반통보를 변환한다.
      DispatchMessage(&msg); // Windows 2000에 조종을 넘긴다.
    }
  }

  return msg.wParam;
}

/* 이 함수는 Windows 2000으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받아 들인다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  HDC hdc;
  static char selection[80] = "";
```

```
    NMTREEVIEW *nmptr;
    PAINTSTRUCT paintstruct;
    int i, response;
    RECT WinDim;

    switch(message) {
      case WM_CREATE:
        // 어미창문의 크기를 얻는다.
        GetClientRect(hwnd, &WinDim);

        // 나무구조보기조종체를 작성한다.
        hTreeWndCtrl = CreateWindow(
                        WC_TREEVIEW,
                        "",
                        WS_VISIBLE | WS_TABSTOP | WS_CHILD |
                        TVS_HASLINES | TVS_HASBUTTONS |
                        TVS_LINESATROOT,
                        0, 0, 200, 200,
                        hwnd,
                        NULL,
                        hInst,
                        NULL
                       );

        InitTree( );
        break;
      case WM_COMMAND:
        switch(LOWORD(wParam)) {
          case IDM_EXPAND:
            TreeView_Expand(hTreeWndCtrl, hTreeCurrent,
                            TVE_EXPAND);
            break;
          case IDM_EXPANDALL:
            for(i=0; i<NUM; i++)
              TreeView_Expand(hTreeWndCtrl, hTreeWnd[i],
                              TVE_EXPAND);
            break;
          case IDM_COLLAPSE:
```

```
        TreeView_Expand(hTreeWndCtrl, hTreeCurrent,
                        TVE_COLLAPSE);
        break;
      case IDM_EXIT:
        response = MessageBox(hwnd, "Quit the Program?",
                              "Exit", MB_YESNO);
        if(response == IDYES) PostQuitMessage(0);
        break;
      case IDM_HELP:
        MessageBox(hwnd, "Try the Tree View",
                   "Help", MB_OK);
        break;
    }
    break;
  case WM_NOTIFY:
    nmptr = (LPNMTREEVIEW) lParam;
    if(nmptr->hdr.code == TVN_SELCHANGED) {
      InvalidateRect(hwnd, NULL, 1);
      if(nmptr->itemNew.hItem == hTreeWnd[0])
        strcpy(selection, "Physics.");
      else if(nmptr->itemNew.hItem == hTreeWnd[1])
        strcpy(selection, "Mechanics.");
      else if(nmptr->itemNew.hItem == hTreeWnd[2])
        strcpy(selection, "Electricity.");
      else if(nmptr->itemNew.hItem == hTreeWnd[3])
        strcpy(selection, "Momentum.");
      else if(nmptr->itemNew.hItem == hTreeWnd[4])
        strcpy(selection, "Linear Momentum.");
      else if(nmptr->itemNew.hItem == hTreeWnd[5])
        strcpy(selection, "Angular Momentum.");

      hTreeCurrent = nmptr->itemNew.hItem;
    }
    break;
  case WM_PAINT:
    hdc = BeginPaint(hwnd, &paintstruct);
    report(hdc, selection);
    EndPaint(hwnd, &paintstruct);
```

```
        break;
      case WM_DESTROY: // 프로그람을 끝낸다.
        PostQuitMessage(0);
        break;
      default:
        /* 이 switch 문에서 지정된것 이외의 통보문은
           Windows 2000 에 처리를 맡긴다. */
       return DefWindowProc(hwnd, message, wParam, lParam);
    }
    return 0;
}

// 선택된 항목을 표시한다.
void report(HDC hdc, char *s)
{
    char str[80];

    if(*s) {
      strcpy(str, "Selection is ");
      strcat(str, s);
    }
    else strcpy(str, "No selection has been made.");
    TextOut(hdc, 0, 200, str, strlen(str));
}

// 나무의 목록을 초기화한다.
void InitTree(void)
{
    TVINSERTSTRUCT tvs;
    TVITEM tvi;

    tvs.hInsertAfter = TVI_LAST; // 추가된 순서로 항목들을 배렬한다.
    tvi.mask = TVIF_TEXT;

    tvi.pszText = "Physics";
    tvs.hParent = TVI_ROOT;
    tvs.item = tvi;
    hTreeWnd[0] = TreeView_InsertItem(hTreeWndCtrl, &tvs);
```

```
    hTreeCurrent = hTreeWnd[0];

    tvi.pszText = "Mechanics";
    tvs.hParent = hTreeWnd[0];
    tvs.item = tvi;
    hTreeWnd[1] = TreeView_InsertItem(hTreeWndCtrl, &tvs);

    tvi.pszText = "Electricity";
    tvs.item = tvi;
    tvs.hParent = hTreeWnd[0];
    hTreeWnd[2] = TreeView_InsertItem(hTreeWndCtrl, &tvs);

    tvi.pszText = "Momentum";
    tvs.item = tvi;
    tvs.hParent = hTreeWnd[1];
    hTreeWnd[3] = TreeView_InsertItem(hTreeWndCtrl, &tvs);

    tvi.pszText = "Linear";
    tvs.item = tvi;
    tvs.hParent = hTreeWnd[3];
    hTreeWnd[4] = TreeView_InsertItem(hTreeWndCtrl, &tvs);

    tvi.pszText = "Angular";
    tvs.item = tvi;
    tvs.hParent = hTreeWnd[3];
    hTreeWnd[5] = TreeView_InsertItem(hTreeWndCtrl, &tvs);
  }
```

이 프로그람은 아래의 자원파일을 필요로 한다.

```
#include <windows.h>
#include "tree.h"

TreeViewMenu MENU
{
  POPUP "&Options" {
    MENUITEM "&Expand One\tF2", IDM_EXPAND
    MENUITEM "Expand &All\tF3", IDM_EXPANDALL
```

```
    MENUITEM "&Collapse\tF4", IDM_COLLAPSE
    MENUITEM "E&xit\tCtrl+X", IDM_EXIT
  }
  MENUITEM "&Help", IDM_HELP
}

TreeViewMenu ACCELERATORS
{
  VK_F1, IDM_HELP, VIRTKEY
  VK_F2, IDM_EXPAND, VIRTKEY
  VK_F3, IDM_EXPANDALL, VIRTKEY
  VK_F4, IDM_COLLAPSE, VIRTKEY
  "^X", IDM_EXIT
}
```

머리부파일 TREE.H 의 내용을 아래에 보여 주었다.

```
#define IDM_EXPAND         100
#define IDM_EXPANDALL      101
#define IDM_COLLAPSE       102
#define IDM_EXIT           103
#define IDM_HELP           104
```

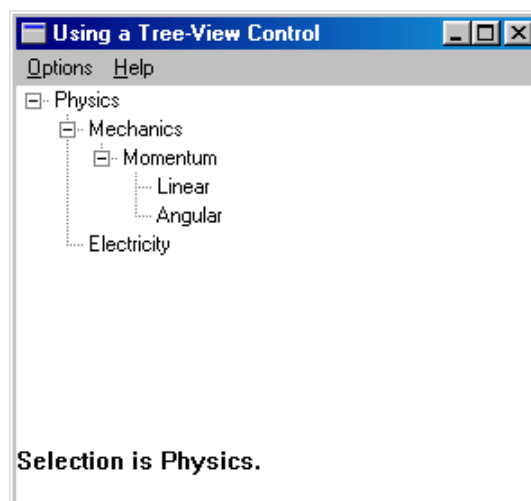



그림 12-4. 나무구조보기조종체 프로그람의 실행결과

이 프로그람에서는 InitTree( )라는 함수에서 나무구조보기조종체를 초기화하고 있다. 매 항목의 손잡이가 hTreeWnd 라는 배렬에 보관되여 있는 점에 주의해야 한다. 이 손잡이들은 나무목록에서 선택된 항목을 식별하는데 사용된다.

hTreeCurrent 는 현재 선택되여 있는 항목을 식별하기 위한 손잡이이다. 이 손잡이는 사용자가 차림표를 사용하여 가지를 전개하거나 접는 때 사용된다.

WindowFunc( )의 내부에서는 새로운 항목이 선택되였을 때 발송되는 WM_NOTIFY 통보문을 처리하고 있다.

itemNew 의 값과 배렬 hTreeWnd 에 보관된 항목의 손잡이목록을 비교하고 있다. 일치하는 손잡이가 있으면 새롭게 선택된 항목을 창문에 표시한다.

이 실례프로그람에서는 나무구조보기조종체의 기본적인 사용방법만을 보여 주며 전체 기능의 일부밖에 사용하지 않고 있다. 나무구조보기조종체를 사용하여 한 나무로부터 다른 나무에 항목을 끌어다 놓는것같은 고급한 기능은 자체로 알아 보아야 한다.

다시 한보 전진

나무구조보기조종체의 표식의 편집(Label editing)

나무안에 표시된 표식의 내용을 변경할수 있는 나무구조보기조종체를 작성할 수도 있다. 그러자면 나무구조보기조종체를 작성할 때의 형식에 TVS_EDITLABELS 를 포함시킨다.

표식의 변경은 TVN_BEGINLABELEDIT 및 TVN_ENDLABELEDIT 라는 두개의 통지문에 의해서 통지된다.(통지문은 WM_NOTIFY 통보문과 함께 발송된다는데 주의해야 한다.)

두 통보문에 있어서 lParam 에는 아래에 보여 준 NMTVDISPINFO 구조체의 지시자가 보관되여 있다.

```
typedef struct tagNMTVDISPINFO {
    NMHDR hdr;
    TVITEM item;
}   NMTVDISPINFO;
```

사용자가 표식의 변경을 시작하면 TVN_BEGINLABELEDIT 통지문이 발송된다. 사용자에게 표식의 편집을 허가하려면 프로그람에서 령을 돌려 준다. 편집을 금지하려면 령 아닌 값을 돌려 준다.

사용자가 편집을 끝내면 TVN_EDNLABELEDIT 통지문이 발송된다. 사용자가 편집을 취소한 경우에는 Item 의 pszText 성원에 NULL 이 보관된다. 그렇지 않은 경우에는 pszText 성원에 새로운 표식의 지시자가 보관된다. 편집이 취소된 경우에는 프로그람에서 령을 돌려 주어 본래의 표식으로 복귀할수 있다. 그렇지

않은 경우에는 령 아닌 값을 돌려 주며 새로운 표식이 사용된다.

실례프로그람을 개조하여 표식을 변경하는 기능을 시험해 보기 위해 나무구조보기조종체를 작성할 때의 형식에 TVS_EDITLABELS 를 추가하고 WindowsFunc( )의 WM_NOTIFY 의 case 문을 아래의 프로그람코드와 치환해 본다.

```
case WM_NOTIFY:
   nmptr = (LPNMTREEVIEW) lParam;
   switch(nmptr->hdr.code) {
     case TVN_BEGINLABELEDIT:
       return 0;
     case TVN_ENDLABELEDIT:
       if(((NMTVDISPINFO *) nmptr)->item.pszText)
         return 1; // 표식이 편집되였다.
       else
         return 0; // 사용자가 취소했다.
   }
   break;
```

이러한 변경을 진행하면 나무구조보기조종체의 표식을 변경할수 있게 된다.

제 13 장

특성표와 조수

이 장에서는 Windows 2000 의 가장 흥미 있는 조종체의 하나인 *조수*(Wizard)의 작성방법에 대해 설명한다. 조수는 복잡한 사용자입력을 인도하기 위해 련속적으로 표시되는 대화칸들의 모임이다. 조수를 구성하는 대화칸들은 조수에 의해 표시되는 순서로 조작된다. Windows 2000 에서는 많은 장면들에서 조수가 사용된다. 례를 들어 새로운 인쇄기를 설치할 때는 인쇄기추가조수가 사용된다.

후에 알게 되지만 조수는 특성표(Property sheet)라고 부르는 공통조종체를 사용하여 작성된다. 특성표는 어떤 항목의 속성값을 표시하거나 속성값을 설정하는데 사용된다. 앞장에서 설명한 표쪽조종체와 형태적으로 류사하지만 특성표가 보다 강력한 기능을 가지고 있다.

특성표가 조수의 기초로 되는것만큼 우선 특성표에 대한 설명으로부터 시작해 보자.

특성표의 기초

특성표(Property sheet)는 사용자에게 어떤 항목과 관련된 여러가지 속성들을 표시하거나 속성을 설정하도록 하기 위한것이다. 실례로 인쇄기나 모뎀의 설정 등에 특성표가 리용되고 있다. 사용자의 시점에서 보면 특성표는 한개이상의 페지들의 집합체이다. 매 페지에는 표쪽이 붙어 있다. 표쪽을 선택하면 그 페지가 능동상태로 된다. 그림 13-1에 특성표의 실례를 보여 주었다.

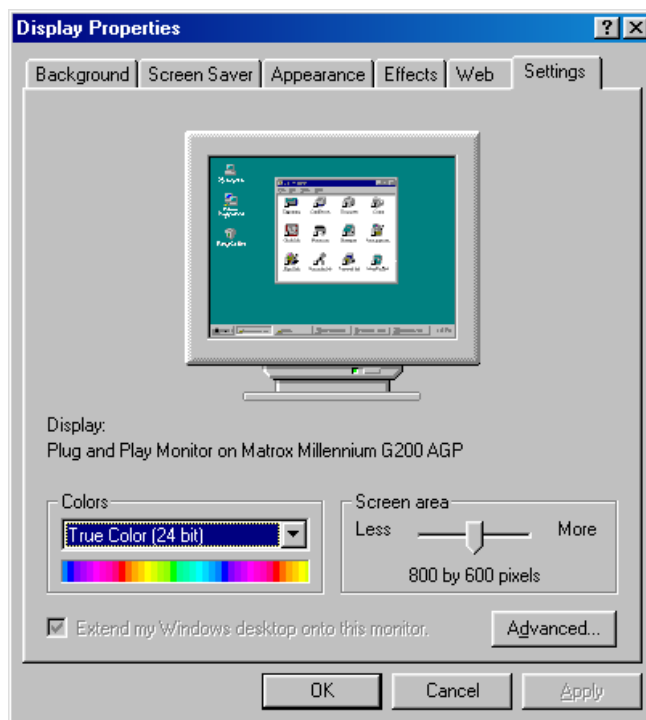



그림 13-1. 특성표의 례

프로그람작성자의 시점에서 보면 특성표는 한개이상의 *비양식화대화칸*의 집합체이다. 특성표의 매 페지는 대화칸본보기(Dialog box template)에서 정의되고 그의 처리는 대화함수에서 진행된다. 매 대화칸본보기는 응용프로그람의 자원파일에서 정의된다.

모든 특성표에는 [OK]와 [Cancel]의 두 단추가 있다. [Apply]라는 세번째 단추가 있는 경우도 종종 있다. 또한 [Help]단추가 있는 경우도 있다.

매 페지와 관련된 대화함수가 사용자에게 속성을 표시하거나 설정하도록 하는 기능을 제공하고 있지만 사용자의 조작을 얻거나 취소하는 기능은 특성표조종체만이 제공한

다. 달리 말한다면 매 페지의 대화함수에는 [OK]나 [Cancel]단추가 들어 있지 않다는 것이다. 이 두 단추는 특성표조종체에 의해 제공된다.

특성표를 구성하는 대화칸은 특성표조종체의 한 부분으로 된다. 특성표조종체는 매 페지 및 페지사이의 련관을 관리한다. 매 대화함수는 그 안의 조종체들을 보통 때처럼 관리한다. 다시말하여 매 페지의 조종체들은 그 페지의 대화함수에서 표준적인 수법으로 조작할수 있다. 그러나 매 페지는 그것이 속해있는 특성표로부터 보내 오는 통보문에도 응답할 의무가 있다. 특성표가 매 페지와 통신을 취하는 경우에는 WM_NOTIFY 통보문을 전송한다. 특성표의 매 페지는 이 통보문에 응답하여야 한다.(자세한 처리내용에 대해서는 뒤에서 설명한다.)

특성표를 작성할 때 지켜야 할 중요한 규칙이 하나 있다. 그것은 매 페지와 련관된 비양식화대화칸이 자기자체를 닫아서는 안된다는것이다. 이것은 DestroyWindow( )를 호출해서는 안된다는 의미이다. 그대신 특성표조종체가 대화칸을 닫는 처리를 진행한다. 만일 한개 페지의 대화칸을 닫아 버렸다면 그 대화칸만이 비여 있게 되고 만다. 이것은 Windows 2000의 형식에 위반되는것이다.

참고 *: 특성표는 공통조종체로서 제공되므로 프로그람에 COMMCTRL.H를 포함시키고 COMCTL32.LIB를 련결하여야 한다.*

특성표의 작성

특성표를 작성하려면 아래와 같은 네가지 단계가 필요하다.

- PROPSHEETPAGE 구조체에 매 페지의 정보를 설정한다.
- CreatePropertySheetPage( )를 호출하여 매 페지를 작성한다.
- PROPSHEETHEADER 구조체에 특성표자체의 정보를 설정한다.
- PropertySheet( )를 호출하여 특성표를 작성하거나 표시한다.

그러면 매 단계를 설명해 보자.

특성표의 페지정의

특성표의 매 페지는 *PROPSHEETPAGE 구조체*에 정의하여야 한다. PROPSHEETPAGE 구조체를 아래에 보여 주었다.

```
typedef struct _PROPSHEETPAGE {
    DWORD dwSize;
    DWORD dwFlags;
    HINSTANCE hInstance;
    union {
            LPCSTR pszTemplate;
            LPCDLGTEMPLATE pResource;
    };
    union {
            HICON hIcon;
            LPCSTR pszIcon;
    };
    LPCSTR pszTitle;
    DLGPROC pfnDlgProc;
    LPARAM lParam;
    LPFNPSPCALLBACK pfnCallback;
    UINT FAR *pcRefParent;
    LPCSTR pszHeaderTitle;
    LPCSTR pszHeaderSubTitle;
}   PROPSHEETPAGE;
```

dwSize에는 PROPSHEETPAGE구조체의 크기를 byte단위로 설정한다.

dwFlags의 값은 유효한 정보가 들어 있는 성원을 가리킨다. 이 값은 표 13-1에 보여 준 한개이상의 기발들의 조합이여야 한다. 이 기발들은 OR 연산자를 리용하여 조합할수 있다.

표 13-1. PROPSHEETPAGE의 dwFlags성원의 값

기 발	의 미
PSP_DEFAULT	체계설정을 사용한다.
PSP_DLGINDIRECT	pszTemplate가 아니라 pResource를 사용한다.
PSP_HASHELP	[Help]단추를 표시한다.
PSP_HIDEHEADER	조수에 머리부를 표시하지 않는다.(조수의 경우에만)
PSP_PREMATURE	특성표조종체가 작성될 때 페지도 작성한다. 보통은 페지가 제일 처음 능동으로 될 때까지 작성되지 않는다.

PSP_RTLREADING	본문을 오른쪽에서 왼쪽으로 표시한다
PSP_USECALLBACK	pfnCallback 를 유효로 한다.
PSP_USEHEADERTITLE	pszHeaderTitle 을 유효로 한다.(조수의 경우에만)
PSP_USEHEADERSUBTITLE	pszHeaderSubTitle 을 유효로 한다.(조수의 경우에만)
PSP_USEHICON	hIcon 을 유효로 한다.
PSP_USEICONID	pszIcon 을 유효로 한다.
PSP_USEREFPARENT	참조계수기를 유효로 하는 PSP_USETITLE 페지의 대화칸본보기에서 정의된 제목이 아니라 pszTitle 에서 지정된 형식을 사용한다.

hInstance 에는 응용프로그람의 실체손잡이를 설정한다. pszTemplate 에는 페지에 련관된 대화칸본보기의 이름이나 ID 를 설정한다. 그러나 *PSP_DLGINDIRECT* 기발을 설정한 경우에는 pszTemplate 가 무시되고 pResource 에서 지정된 대화칸이 사용된다.

페지의 표쪽에 작은 아이콘을 포함시키려는 경우는 그것을 hIcon 혹은 pszIcon 의 어느 하나에 설정한다. 이 경우에는 적절한 기발도 설정해야 한다. hIcon 에는 아이콘의 손잡이를 설정한다. pszIcon 아이콘에는 자원에서 정의된 아이콘의 이름 혹은 ID를 설정한다.

일반적으로는 페지에 련관된 대화칸의 제목이 그 페지의 제목으로 된다.이 제목은 페지의 표쪽에 표시된다. 그러나 pszTitle 에 새로운 제목으로 될 문자렬의 지시자를 설정하여 다른 제목을 표시할수도 있다. 이 경우에는 *PSP_USETITLE* 기발을 설정하여야 한다.

페지에 련관된 비양식화대화칸의 대화함수를 pfnDlgProc 에 설정한다.

lParam 에는 응용프로그람 자체의 자료를 설정할수 있다.

pfnCallback 를 유효로 한 경우에는 페지가 작성되거나 파괴될 때 호출되는 *역호출함수*를 지정할수 있다. 이 함수는 이 장의 실례프로그람에서는 요구되지 않지만 자체로 작성하는 응용프로그람에서 이 함수가 필요한 경우에는 다음과 같은 방법으로 선언하여야 한다.

```
UINT CALLBACK PropPageFunc(HWND hwnd,UINT message,
                           LPPROPSHEETPAGE lpPropSheet);
```

이 함수가 호출될 때는 hwnd 값이 NULL 로 된다. messge 의 값은 PSPCB_ADDREF (페지가 참조되려고 하고 있다), PSPCB_CREATE (페지가 작성되려고 하고 있다) 또는 PSPCD_RELEASE (페지가 해제되려고 하고 있다)중의 어느 하나로 된다. lpPropSheet 는 대상으로 되는 페지의 PROPSHEETPAGE 구조체의 지시자이다.

message 의 값이 PSPCB_CREATE 인 경우는 함수에서 령 아닌 값을 돌려 주어 작성을 실행하고 령을 돌려 주어 페지의 작성을 취소할수 있다. message 의 값이 PSPCB_ADDREF 혹은 PSPCD_RELEASE 인 경우는 돌림값이 사용되지 않는다.

pcRefParent 에는 참조계수기로 되는 변수의 주소를 설정한다. 이 성원은 PSP_USEREFPARENT 가 설정된 경우에만 유효하다.

pszHeaderTitle 와 pszHeaderSubTitle 은 조수인 경우에만 유효하게 된다. 이 성원들에는 제목 혹은 부분제목으로 되는 문자렬의 지시자를 설정한다. 이 문자렬들은 조수의 머리부령역에 표시된다. dwFlags 의 값에 PSP_USEHEADERTITLE 혹은 PSP_USEHEADERSUBTITLE 이 포함되여 있지 않는 경우는 이 두개의 성원들이 무시된다.

이식과 관련한 요점: 이전의 PROPSHEETPAGE 구조체에는 pszHeaderTitle, pszHeaderSubTitle 및 그것들과 관계되는 기발들이 들어 있지 않다.

매 페지의 초기화

PROPSHEETPAGE 구조체에 필요한 정보를 설정하고 나서 *CreatePropertySheetPage( )*를 호출하여 페지를 작성한다. 선언은 다음과 같다.

```
HPROPSHEETPAGE CreatePropertySheetPage(
                                    LPCPPROPSHEETPAGE lpPage);
```

lpPage 에는 PROPSHEETPAGE 구조체의 지시자를 설정한다. 이 함수는 새롭게 작성된 페지의 손잡이를 돌려 준다. 이 손잡이는 후에 특성표를 작성할 때 필요하므로 보관해 둔다. 페지를 작성할수 없는 경우에는 함수의 돌림값이 NULL 로 된다.

PROPSHEETHEADER 구조체의 초기화

매 페지의 작성이 끝나면 PROPSHEETHEADER 구조체에 특성표의 초기화정보를 설정한다. PROPSHEETHEADER 구조체의 정의를 아래에 보여 주었다.

```
typedef struct _PROPSHEETHEADER {
    DWORD dwSize;
    DWORD dwFlags;
    HWND hwndParent;
    HINSTANCE hInstance;
    union {
```

```
                HICON hIcon;
                LPCSTR pszIcon;
        };
        LPCSTR pszCaption;
        UINT nPages;
        union {
                UINT nStartPage;
                LPCSTR pStartPage;
        };
        union {
                LPCPROPSHEETPAGE ppsp;
                HPROPSHEETPAGE FAR *phpage;
        };
        PFNPROPSHEETCALLBACK pfnCallback;
        union {
                HBITMAP hbmWatermark;
                LPCSTR pszbmWatermark;
        };
        HPALETTE hplWatermark;
        union {
                HBITMAP hbmHeader;
                LPCSTR pszbmHeader;
        };
    }   PROPSHEETHEADER;
```

dwSize 에는 PROPSHEETHEADER 구조체의 크기를 byte 단위로 설정한다.

dwFlags 의 값은 유효한 정보가 보관되여 있는 성원을 가리키는 값이다. 이 값은 표 13-2 에 보여 준 한개이상의 기발의 조합이여야 한다. 이 기발들은 OR 연산자를 사용하여 조합할수 있다.

표 13-2. PROPSHEETHEADER 의 dwFlags 성원의 값

기　발	의　미
PSH_DEFAULT	체계설정을 사용한다.
PSH_HASHELP	[Help] 단추를 유효로 한다.
PSH_HEADER	머리부에 비트매프를 표시한다.(조수의 경우에만)

PSH_MODELESS	비양식화특성표조종체를 작성한다. 체계설정으로 특성표조종체는 양식화로 된다.
PSH_NOAPPLYNOW	[Apply] 단추를 표시하지 않는다.
PSH_NOCONTEXTHELP	상황의존도움말의 ?를 표시하지 않는다.(특성표의 경우에만)
PSH_PROPSHEETPAGE	ppsp 를 유효로 하고 phpage 를 무효로 한다.
PSH_PROPTITLE	pszCaption 에 지정된 제목에 [Property of]라는 문자렬을 추가한다.
PSH_RTLREANDING	본문을 오른쪽에서 왼쪽으로 표시한다.
PSH_USECALLBACK	pfnCallback 을 유효로 한다.
PSH_USEHBMHEADER	hbmHeader 를 유효로 한다. 그렇지 않은 경우는 pszbmHeader 가 사용된다.(조수의 경우에만)
PSH_USEHBMWATERMARK	hbmWaterMark 를 유효로 한다. 그렇지 않은 경우는 pszbmWaterMark 가 사용된다.(조수의 경우에만)
PSH_USEHICON	hIcon 을 유효로 한다.
PSH_USEICONID	pszIcon 을 유효로 한다.
PSH_USEHPLWATERMARK	hplWaterMark 를 유효로 한다.(조수의 경우에만)
PSH_USEPAGELANG	사용되는 언어가 선두페지에 정의된것이라는것을 가리킨다.
PSH_USEPSTARTPAGE	pStartPage 를 유효로 하고 nStartPage 를 무효로 한다.
PSH_WATERMARK	투명도안을 포함한다.(조수의 경우에만)
PSH_WIZARD	조수를 작성한다.
PSH_WIZARD97	머리부의 제목과 부문제목, 투명도안 및 비트매프를 포함한 조수를 작성한다. 이것은 Windows 2000 조수에서 사용되는 형식이다.
PSH_WIZARDHASFINISH	조수의 모든페지에 [Finish]단추를 포함한다.(조수의 경우에만)
PSH_WIZARD_LITE	PSH_WIZARD97 에 류사한 간단한 조수를 작성한다.

hwndParent 에는 특성표의 어미창문의 손잡이를 설정한다. hInstance 에는 응용프로그람의 실체손잡이를 설정한다.

특성표의 제목띠에 작은 아이콘을 표시하려는 경우에는 그것을 hIcon 혹은 pszIcon 에 설정한다. 이 경우에는 적절한 기발설정도 필요하게 된다. hIcon 에는 아이콘의 손잡이를 설정한다. pszIcon 에는 자원파일에 정의되여 있는 아이콘의 이름 혹은 ID 를 설정한다.

특성표조종체의 창문제목을 pszCaption 에 설정한다. 특성표의 페지수를 nPages 에 설정한다.

특성표가 능동상태로 되였을 때 제일 먼저 표시되는 페지의 색인을 nStartPage 혹은 pStartPage 의 어느 하나에 설정한다. 체계설정으로는 nStartPage 가 사용된다. nStartPage 에는 첫 페지의 색인을 설정한다. 페지의 색인값은 령으로부터 시작한다. PSH_USEPSTARTPAGE 기발을 설정한 경우는 pStartPage 에 페지의 이름 혹은 ID 를 설정하여야 한다.

phpage 에는 특성표의 매 페지의 손잡이를 보관한 배렬의 지시자를 보관한다. 이 손잡이들은 앞에서 설명한 *CreatePropertySheetPage( )*를 사용하여 작성된다. 그러나 PSH_PROPSHEETPAGE 기발을 설정한 경우에는 이 방법이 아니라 ppsp 에 PROPSHEETPAGE 구조체배렬의 주소를 설정하여야 한다. 이 경우에는 손잡이가 자동적으로 작성되므로 *CreatePropertySheetPage( )*를 호출할 필요가 없다. 다만 다른 처리에서도 페지의 손잡이를 사용하는 경우가 있으므로 손잡이를 명시적으로 작성하는 방법을 사용하는편이 유익하다.

pfnCallback 가 유효한 경우에는 거기에 특성표가 작성되였을 때 호출되는 역호출함수를 설정한다. 이 장의 실례프로그람에서는 사용되지 않지만 이러한 역호출함수가 필요하게 되는 경우도 있다. 이 역호출함수는 다음과 같이 선언하여야 한다.

```
int CALLBACK PropSheetFunc(HWND hdwnd, UINT message,
                           LPARAM lParam);
```

이 함수가 호출되였을 때 hdwnd 에는 특성표조종체의 손잡이가 보관된다. message 의 값은 PSCB_INITIALIZED(특성표가 초기화되려고 하고 있다.) 또는 PSCB_PRECREATE(특성표가 작성되려고 하고 있다.)의 어느 한 값으로 된다. PSCB_INITIALIZED 의 경우에는 lParam 에 령이 보관된다. PSCB_PRECREATE 의 경우에는 lParam 에 대화칸본보기의 지시자가 보관되고 hdwnd 에 NULL 이 보관된다. 어느 경우에도 함수의 돌림값으로서 령을 돌려 주어야 한다.

조수에서는 *투명도안*과 *머리부비트매프*를 설정할수 있다. 투명도안(Watermark)은 페지의 왼쪽에 표시되는 비트매프이다. 머리부비트매프(Header bitmap)는 조수의 머리부의 오른쪽에 표시된다. 이 비트매프들을 표시하려면 PSH_WIZARD97 을 포함하여 적절한 기발을 설정하여야 한다. PSH_WIZARD97 은 Windows 2000 에서 주장되고 있는 형식이므로 Windows 2000 의 모든 조수들은 이러한 비트매프들을 사용해야 한다.

hbmWatermark 에 투명도안으로 사용되는 비트매프의 손잡이를 설정한다.

pszbmWatermark 를 사용하여 비트매프의 이름을 설정할수도 있다. 일반적으로는 체계설정의 조색판을 사용하여 투명도안이 그려 지지만 hplWatermark 에 조색판의 손잡이를 설정할수도 있다.어느 경우에나 적절한 기발을 설정하여야 한다.

머리부비트매프의 손잡이를 hbmHeader 에 설정한다. pszbmHeader 에는 머리부의 자원이름을 설정할수도 있다. 이 경우도 적절한 기발을 설정해야 한다.

이식과 관련한 요점 : 이전의 PROPSHEETHEADER 에는 hbmWatermark, pszbmWatermark, hplWatermark, hbmHeader 및 pszbmHeader 가 없다. 이 성원들을 유효하게 하는 기발도 정의되여 있지 않다.

특성표조종체의 작성

페지의 정의와 *PROPSHEETHEADER 구조체*의 정의가 끝나면 *특성표조종체*를 작성할수 있다. 그러자면 *PropertySheet( )*함수를 사용한다. 아래에 선언을 보여 주었다.

```
int PropertySheet(LPCPPROPSHEETHEADER lpHeader);
```

lpHeader 에는 PROPSHEETHEADER 구조체의 지시자를 설정한다. 호출이 성공하면 정의값이 돌려 지고 실패하면 부의값(-1)이 돌려 진다. 비양식화특성표조종체를 작성할 때는(PSH_MODELESS 기발을 설정한 경우) 특성표의 손잡이가 돌려 진다.

특성표통보문의 처리

이미 설명한바와 같이 특성표조종체는 WM_NOTIFY 통보문을 사용하여 매 페지의 대화함수에 *통지문*을 보낸다. 대부분의 통지문에서 lParam 에는 아래에 보여 주는 *PSHNOTIFY* 구조체의 지시자가 보관된다.

```
typedef struct _PSHNOTIFY {
    NMHDR hdr;
    LPARAM lParam;
}   PSHNOTIFY;
```

첫 성원인 *NMHDR 구조체*는 통지문의 내용을 가리킨다. NMHDR 구조체의 정의는 다음과 같다.

```
typedef struct tagNMHDR
```

```
{
    HWND hwndFrom;
    UINT idFrom;
    UINT code;
}   NMHDR;
```

특성표의 WM_NOTIFY 통보문의 경우에는 hwndFrom 에 특성표조종체의 손잡이가 보관된다. code 의 값은 조작내용을 알려 주는 통지문으로 된다. PSHNOTIFY 구조체의 lParam 은 모든 통보문에서 사용되는것은 아니다. 그것이 사용되는 경우에만 그 속에 통보문의 보충정보가 보관된다. 이 장의 실례프로그람에서 lParam 의 값은 사용되지 않는다. hdr 의 정보만이 필요로 된다.

특성표조종체는 매 대화칸에서 발생한 여러가지 사건을 알려 주기 위하여 통지문을 사용한다. 례하면 페지가 선택된 때, 특성표의 단추가 눌리워 진 때, 또는 페지가 변경된 때 등에서 통지문이 발송된다.

통지문은 아래조건을 정확히 알려 주는것이다. 주요한 통지문을 표 13-3 에 주었다.

표 13-3. 특성표의 주요통지문

통지문	의　　미	돌림값
PSN_APPLY	사용자가 [Apply]또는 [Ok]단추를 눌렀다.	변경을 허가하는 경우는 PSNRET_NOERROR, 변경을 허가하지 않는 경우는 PSNRET_INVALID_NOCHANGEPAGE
PSN_HELP	사용자가 [Help]단추를 눌렀다.	없음
PSN_KILLACTIVE	페지가 초점을 잃었거나 [OK]단추를 눌렀다.	비능동으로 하는것을 허가하는 경우 령, 허가하지 않는 경우 령 아닌 값
PSN_QUERYCANCEL	사용자가 [Cancel]단추를 눌렀다.	Cancel 을 허가하는 경우는 령, 허가하지 않는 경우는 령 아닌 값
PSN_RESET	사용자가 [Cancel]단추를 눌렀다.	없음
PSN_SETACTIVE	페지가 초점을 가졌다.(능동으로 되였다.)	능동으로 한다는것을 허가하는 경우 령, 그렇지 않은 경우는 능동으로 하는 페지의 ID

PSN_WIZBACK	사용자가 [Back]단추를 눌렀다.(조수의 경우에만)	앞페지를 능동으로 하는 경우는 령, 그렇지 않는 경우는 -1 또는 능동으로 하는 페지의 ID
PSN_WIZFINISH	사용자가 [Finish]단추를 눌렀다.(조수의 경우에만)	조수를 완료하는 경우 령, 그렇지 않는 경우는 령 아닌 값
PSN_WIZNEXT	사용자가 [Next]단추를 눌렀다.	다음 페지를 능동으로 하는 경우에는 령, 그렇지 않은 경우는 -1 또는 능동으로 하는 페지의 ID

일부 상황에서는 대화함수가 특성표조종체에 값을 돌려 주어 통지문에 응답한다. 그렇게 하려면 *SetWindowLong( )*함수를 사용한다. 이 함수는 창문과 관련된 여러가지 속성들을 설정한다. SetWindowLong( )함수의 선언을 아래에 보여 주었다.

```
LONG SetWindowLong(HANDLE hwnd, int index,
                   LONG value);
```

hwnd 에는 대화칸의 손잡이를 설정한다. index 에는 설정할 속성의 종류를 설정한다. 특성표에 값을 돌려 주는 경우에는 index에 DWL_MSGRESULT를 설정한다. 실제로 돌려 주는 값은 value 에 설정한다. 대화함수에서 통보문을 처리한 때는 TRUE, 처리하지 못한 때는 FALSE를 돌려 준다.

SetWindowLong( )함수를 사용하여 돌림값을 돌려 주는 수법은 WM_NOTIFY 통보문의 경우에만 사용된다.

특성표에 통보문을 보내기

통보문을 받을뿐만아니라 응용프로그람이 특성표조종체에 통보문을 보낼수도 있다. 이를 위하여 통보문을 받는 측으로 특성표조종체의 손잡이를 지정하여 SendMessage( )를 사용한다. 특성표에 보내는 모든 통보문은 PSM_이라는 앞붙이로 시작한다. 주요한 통보문들을 표 13-4 에 주었다.

표 13-4. 특성표의 주요통보문

통 보 문	의　　미
PSM_APPLY	PSN_APPLY 통보문을 보낸다. lParam 과 wParam 에 령을 설정한다. 호출이 성공하면 령아닌 값이 돌려 지고 실패하면 령이 돌려 진다.
PSM_CHANGED	[Apply]단추를 유효로 한다. lParam 에 령을 설정한다. wParam 에는 페지의 대화칸의 손잡이를 설정한다. 돌림값은 돌려 지지 않는다.
PSM_SETCURSEL	페지를 변경한다. 새로운 페지의 손잡이를 설정한다. wParam 에는 새로운 페지의 색인을 설정한다. 호출이 성공하면 령 아닌 값이 돌려 지고 실패하면 령이 돌려 진다.
PSM_SETWIZBUTTONS	조수의 단추를 유효로 한다.(조수의 경우에만) lParam 에는 PSWWIZ_BACK, PSWWIZ_NEXT, PSWWIZ_FINISH 및 PSWWIZ_DISABLEFINISH 의 어느 한 기발을 설정한다. wParam 에는 령을 설정한다. 돌림값은 돌려 지지 않는다.
PSM_UNCHANGED	[Apply]단추를 무효로 한다. lParam 에는 령을 설정한다. wParam 에는 페지의 대화칸의 손잡이를 설정한다. 돌림값은 돌려 지지 않는다.

Windows 2000 은 특성표에 통보문을 보내는 조작을 간단히 실현할수 있는 마크로들을 제공하고 있다. 표 13-4 에 대응되는 마크로들을 아래에 주었다.

```
BOOL PropSheet_Apply(hPropSheet);
BOOL PropSheet_Changed(hPropSheet, hPageDialog);
BOOL PropSheet_SetCurSel(hPropSheet, hPage, index);
BOOL PropSheet_SetWizButtons(hPropSheet, Flags);
BOOL PropSheet_Unchanged(hPropSheet, hPageDialog);
```

hPropSheet 에는 통보문을 보내는 특성표조종체의 손잡이를 설정한다. hPageDialog 에 통보문을 보내는 페지의 대화함수손잡이를 설정한다. Index 에 다음에 선택될 페지의 색인을 설정한다. hPage 에 페지의 손잡이를 설정한다. Flags 에 유효로 할 조수의 단추를 설정한다. *PropSheet_SetWizButtons()*마크로는 조수의 경우에만 사용된다. 상세한 내용은 이 장의 뒤부분에서 설명한다.

PSM_CHANGED 통보문은 [Apply]단추를 유효로 하는것으로서 특히 중요한 통보문

이다. 사용자가 조작판에 어떤 변경을 진행한 경우에는 응용프로그람에서 특성표조종체에 반드시 이 통보문을 보내야 한다.

특성표대화칸의 크기

특성표의 토대로 되는 대화칸은 임의의 크기로 할수 있다. 그러나 특성표의 크기는 일반적인 관례를 지켜 표준적인 크기로 설정하는것이 좋다. 표준적인 크기에는 다음과 같은것들이 있다.

크기	값	높이
소형	PROP_SM_CXDLG	PROP_ SM_CYDLG
중형	PROP_MED_CXDLG	PROP_ MED_CYDLG
대형	PROP_LG_CXDLG	PROP_ LG_CYDLG

이 마크로들은 COMMCTRL.H 에 정의되여 있다. 크기는 대화단위로 정의되여 있으며 응용프로그람의 자원파일의 DIALOG 명령부분에서 사용된다.

특성표의 실례프로그람

실례 13-1 에 보여 준 프로그람은 세개의 페지를 가진 특성표를 표시하는 실례이다. 이 특성표에서는 실제로 어떠한 속성도 설정하지 않지만 특성표를 작성하거나 표시하기 위해 필요한 절차들을 파악할수 있을것이다.

실례 13-1. Prop 프로그람

```
// 특성표의 실례프로그람

#include <windows.h>
#include <cstdio>
#include <commctrl.h>
#include "prop.h"

#define NUMSTRINGS 5
#define NUMPAGES 3

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK DialogFunc1(HWND, UINT, WPARAM, LPARAM);
```

```
BOOL CALLBACK DialogFunc2(HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK DialogFunc3(HWND, UINT, WPARAM, LPARAM);

char szWinName[] = "MyWin"; // 창문클라스의 이름

HINSTANCE hInst;

HWND hDlg; // 대화칸의 손잡이
HPROPSHEETPAGE hPs[NUMPAGES];
HWND hPropSheet;
HWND hPage[NUMPAGES];

char list[][40] = {
  "Windows 2000",
  "Windows NT 4",
  "Windows 98",
  "Windows 95",
  "Windows 3.1"
};

int cb1=0, cb2=0, cb3=0;
int rb1=1, rb2=0, rb3=0;
int lb1sel=0;

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;
  HACCEL hAccel;
  INITCOMMONCONTROLSEX cc;

  // 창문클라스를 정의한다.
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst;      // 실체의 손잡이
  wcl.lpszClassName = szWinName; // 창문클라스의 이름
```

```
wcl.lpfnWndProc = WindowFunc;  // 창문함수
wcl.style = 0;                 // 체계설정의 형식

wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

wcl.lpszMenuName = "PropSheetMenu"; // 기본차림표

wcl.cbClsExtra = 0; // 보조기억기령역은 필요 없다.
wcl.cbWndExtra = 0;

// 창문의 배경색을 흰색으로 한다.
wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

// 창문클라스를 등록한다.
if(!RegisterClassEx(&wcl)) return 0;

/* 창문클라스가 등록되였으므로
   창문을 작성할수 있다. */
hwnd = CreateWindow(
  szWinName, // 창문클라스의 이름
  "Demonstrate a Property Sheet", // 제목
  WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
  CW_USEDEFAULT, // X 자리표는 Windows 가 결정하게 한다.
  CW_USEDEFAULT, // Y 자리표는 Windows 가 결정하게 한다.
  CW_USEDEFAULT, // 너비는 Windows 가 결정하게 한다.
  CW_USEDEFAULT, // 높이는 Windows 가 결정하게 한다.
  NULL,          // 어미창문은 없다.
  NULL,          // 차림표는 없다.
  hThisInst,     // 실체의 손잡이
  NULL           // 추가파라메터는 없다.
);

hInst = hThisInst; // 현재의 실체손잡이를 보관한다.

// 건반가속기를 적재한다.
hAccel = LoadAccelerators(hThisInst, "PropSheetMenu");
```

```
    // 공통조종체를 초기화한다.
    cc.dwSize = sizeof(INITCOMMONCONTROLSEX);
    cc.dwICC = ICC_TAB_CLASSES;
    InitCommonControlsEx(&cc);

    // 창문을 표시한다.
    ShowWindow(hwnd, nWinMode);
    UpdateWindow(hwnd);

    // 통보문순환고리를 작성한다.
    while(GetMessage(&msg, NULL, 0, 0))
    {
      if(!TranslateAccelerator(hwnd, hAccel, &msg)) {
        TranslateMessage(&msg); // 건반통보를 변환한다.
        DispatchMessage(&msg); // Windows 2000 에 조종을 넘긴다.
      }
    }

    return msg.wParam;
}

/* 이 함수는 Windows 2000 으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받아 들인다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
    int response;
    PROPSHEETPAGE PropSheet[NUMPAGES];
    PROPSHEETHEADER PropHdr;

    switch(message) {
      case WM_COMMAND:
        switch(LOWORD(wParam)) {
          case IDM_DIALOG:
            PropSheet[0].dwSize = sizeof(PROPSHEETPAGE);
            PropSheet[0].dwFlags = PSP_DEFAULT;
```

```
        PropSheet[0].hInstance = hInst;
        PropSheet[0].pszTemplate = "PropSheetDB1";
        PropSheet[0].pszIcon = NULL;
        PropSheet[0].pfnDlgProc = (DLGPROC) DialogFunc1;
        PropSheet[0].pszTitle = "";
        PropSheet[0].lParam = 0;
        PropSheet[0].pfnCallback = NULL;

        PropSheet[1].dwSize = sizeof(PROPSHEETPAGE);
        PropSheet[1].dwFlags = PSP_DEFAULT;
        PropSheet[1].hInstance = hInst;
        PropSheet[1].pszTemplate = "PropSheetDB2";
        PropSheet[1].pszIcon = NULL;
        PropSheet[1].pfnDlgProc = (DLGPROC) DialogFunc2;
        PropSheet[1].pszTitle = "";
        PropSheet[1].lParam = 0;
        PropSheet[1].pfnCallback = NULL;

        PropSheet[2].dwSize = sizeof(PROPSHEETPAGE);
        PropSheet[2].dwFlags = PSP_DEFAULT;
        PropSheet[2].hInstance = hInst;
        PropSheet[2].pszTemplate = "PropSheetDB3";
        PropSheet[2].pszIcon = NULL;
        PropSheet[2].pfnDlgProc = (DLGPROC) DialogFunc3;
        PropSheet[2].pszTitle = "";
        PropSheet[2].lParam = 0;
        PropSheet[2].pfnCallback = NULL;

        hPs[0] = CreatePropertySheetPage(&PropSheet[0]);
        hPs[1] = CreatePropertySheetPage(&PropSheet[1]);
        hPs[2] = CreatePropertySheetPage(&PropSheet[2]);

        PropHdr.dwSize = sizeof(PROPSHEETHEADER);
        PropHdr.dwFlags = PSH_DEFAULT;
        PropHdr.hwndParent = hwnd;
        PropHdr.hInstance = hInst;
        PropHdr.pszIcon = NULL;
        PropHdr.pszCaption = "Sample Property Sheet";
```

```
          PropHdr.nPages = NUMPAGES;
          PropHdr.nStartPage = 0;
          PropHdr.phpage = hPs;
          PropHdr.pfnCallback = NULL;

          PropertySheet(&PropHdr);
          break;
        case IDM_EXIT:
          response = MessageBox(hwnd, "Quit the Program?",
                                "Exit", MB_YESNO);
          if(response == IDYES) PostQuitMessage(0);
          break;
        case IDM_HELP:
          MessageBox(hwnd, "Try the Property Sheet", "Help", MB_OK);
          break;
      }
      break;
    case WM_DESTROY: // 프로그람을 끝낸다.
      PostQuitMessage(0);
      break;
    default:
      /* 이 switch문에서 지정된것 이외의 통보문은
         Windows 2000에 처리를 맡긴다. */
      return DefWindowProc(hwnd, message, wParam, lParam);
  }

  return 0;
}

// 첫번째 대화함수
BOOL CALLBACK DialogFunc1(HWND hdwnd, UINT message,
                          WPARAM wParam, LPARAM lParam)
{
  static long index;
  int i;
  char str[80];

  switch(message) {
```

```
case WM_NOTIFY:
  switch(((NMHDR *) lParam)->code) {
    case PSN_SETACTIVE: // 페지가 입력초점을 얻었다.
      hPropSheet = ((NMHDR *) lParam)->hwndFrom;
      SetWindowLong(hdwnd, DWL_MSGRESULT, 0);
      index = lb1sel;
      return 1;
    case PSN_KILLACTIVE: // 페지가 입력초점을 잃었다.
      lb1sel = index;
      SetWindowLong(hdwnd, DWL_MSGRESULT, 0);
      return 1;
  }
  break;
case WM_COMMAND:
  switch(LOWORD(wParam)) {
    case IDD_OPTIONS:
      PropSheet_SetCurSel(hPropSheet, hPs[1], 1);
      return 1;
    case IDD_OPTIMIZE:
      PropSheet_SetCurSel(hPropSheet, hPs[2], 2);
      return 1;
   case IDD_LB1: // 목록칸의 LBN_DBLCLK 를 처리한다.
      PropSheet_Changed(hPropSheet, hdwnd);
      // 사용자가 선택을 진행하였는가를 검사한다.
      if(HIWORD(wParam)==LBN_DBLCLK) {
        index = SendDlgItemMessage(hdwnd, IDD_LB1,
                LB_GETCURSEL, 0, 0);  // 색인을 얻는다.
        sprintf(str, "%s", list[index]);

        MessageBox(hdwnd, str, "Selection Made", MB_OK);
      }
      return 1;
  }
  break;
case WM_INITDIALOG: // 목록칸을 초기화한다.
  for(i=0; i<NUMSTRINGS; i++)
    SendDlgItemMessage(hdwnd, IDD_LB1,
               LB_ADDSTRING, 0, (LPARAM)list[i]);
```

```
      // 첫 항목을 선택한다.
      SendDlgItemMessage(hdwnd, IDD_LB1, LB_SETCURSEL, lb1sel, 0);

      return 1;
    }

    return 0;
}

// 두번째 대화함수
BOOL CALLBACK DialogFunc2(HWND hdwnd, UINT message,
                          WPARAM wParam, LPARAM lParam)
{
  switch(message) {
    case WM_NOTIFY:
      switch(((NMHDR *) lParam)->code) {
        case PSN_SETACTIVE:// 페지가 입력초점을 얻었다.
          hPropSheet = ((NMHDR *) lParam)->hwndFrom;
          SetWindowLong(hdwnd, DWL_MSGRESULT, 0);
          return 1;
        case PSN_KILLACTIVE: // 페지가 입력초점을 잃었다.
          cb1 = SendDlgItemMessage(hdwnd, IDD_CB1,
                            BM_GETCHECK, 0, 0);
          cb2 = SendDlgItemMessage(hdwnd, IDD_CB2,
                            BM_GETCHECK, 0, 0);
          cb3 = SendDlgItemMessage(hdwnd, IDD_CB3,
                            BM_GETCHECK, 0, 0);
          SetWindowLong(hdwnd, DWL_MSGRESULT, 0);
          return 1;
      }
      break;
    case WM_COMMAND:
      switch(LOWORD(wParam)) {
        case IDD_CB1:
        case IDD_CB2:
        case IDD_CB3:
          PropSheet_Changed(hPropSheet, hdwnd);
```

```
          return 1;
        case IDD_ALL:
          PropSheet_Changed(hPropSheet, hdwnd);
          SendDlgItemMessage(hdwnd, IDD_CB1, BM_SETCHECK,
                             BST_CHECKED, 0);
          SendDlgItemMessage(hdwnd, IDD_CB2, BM_SETCHECK,
                             BST_CHECKED, 0);
          SendDlgItemMessage(hdwnd, IDD_CB3, BM_SETCHECK,
                             BST_CHECKED, 0);
          return 1;
      }
      break;
    case WM_INITDIALOG: // 검사칸을 초기화한다.
      SendDlgItemMessage(hdwnd, IDD_CB1, BM_SETCHECK, cb1, 0);
      SendDlgItemMessage(hdwnd, IDD_CB2, BM_SETCHECK, cb2, 0);
      SendDlgItemMessage(hdwnd, IDD_CB3, BM_SETCHECK, cb3, 0);
      return 1;
  }

  return 0;
}

// 세번째 대화함수
BOOL CALLBACK DialogFunc3(HWND hdwnd, UINT message,
                          WPARAM wParam, LPARAM lParam)
{
  switch(message) {
    case WM_NOTIFY:
      switch(((NMHDR *) lParam)->code) {
        case PSN_SETACTIVE: // 페지가 입력초점을 얻었다.
          hPropSheet = ((NMHDR *) lParam)->hwndFrom;
          SetWindowLong(hdwnd, DWL_MSGRESULT, 0);
          return 1;
        case PSN_KILLACTIVE: // 페지가 입력초점을 잃었다.
          rb1 = SendDlgItemMessage(hdwnd, IDD_RB1,
                           BM_GETCHECK, 0, 0);
          rb2 = SendDlgItemMessage(hdwnd, IDD_RB2,
                           BM_GETCHECK, 0, 0);
```

```
            rb3 = SendDlgItemMessage(hdwnd, IDD_RB3,
                                  BM_GETCHECK, 0, 0);
            SetWindowLong(hdwnd, DWL_MSGRESULT, 0);
            return 1;
        }
        break;
      case WM_COMMAND:
        switch(LOWORD(wParam)) {
          case IDD_RB1:
          case IDD_RB2:
          case IDD_RB3:
            PropSheet_Changed(hPropSheet, hdwnd);
            return 1;
          case IDD_FASTEST:
            PropSheet_Changed(hPropSheet, hdwnd);
            SendDlgItemMessage(hdwnd, IDD_RB2, BM_SETCHECK, 0, 0);
            SendDlgItemMessage(hdwnd, IDD_RB3, BM_SETCHECK, 0, 0);
            SendDlgItemMessage(hdwnd, IDD_RB1, BM_SETCHECK, 1, 0);
            return 1;
          case IDD_SMALLEST:
            PropSheet_Changed(hPropSheet, hdwnd);
            SendDlgItemMessage(hdwnd, IDD_RB1, BM_SETCHECK, 0, 0);
            SendDlgItemMessage(hdwnd, IDD_RB2, BM_SETCHECK, 0, 0);
            SendDlgItemMessage(hdwnd, IDD_RB3, BM_SETCHECK, 1, 0);
            return 1;
        }
        break;
      case WM_INITDIALOG: // 단일선택단추를 초기화한다.
        SendDlgItemMessage(hdwnd, IDD_RB1, BM_SETCHECK, rb1, 0);
        SendDlgItemMessage(hdwnd, IDD_RB2, BM_SETCHECK, rb2, 0);
        SendDlgItemMessage(hdwnd, IDD_RB3, BM_SETCHECK, rb3, 0);
        return 1;
    }
```

```
  return 0;
}
```

이 프로그람은 아래의 자원파일을 필요로 한다.

```
// 특성표의 대화칸
#include <windows.h>
#include <commctrl.h>
#include "prop.h"

PropSheetMenu MENU
{
  POPUP "&Property Sheet"
  {
    MENUITEM "&Activate\tF2", IDM_DIALOG
    MENUITEM "E&xit\tCtrl+X", IDM_EXIT
  }
  MENUITEM "&Help", IDM_HELP
}

PropSheetMenu ACCELERATORS
{
  VK_F2, IDM_DIALOG, VIRTKEY
  VK_F1, IDM_HELP, VIRTKEY
  "^X", IDM_EXIT
}

PropSheetDB1 DIALOGEX 0, 0, PROP_SM_CXDLG, PROP_SM_CYDLG
CAPTION "Pick Target"
{
  PUSHBUTTON "Options", IDD_OPTIONS, 11, 24, 34, 14
  PUSHBUTTON "Optimize", IDD_OPTIMIZE, 11, 48, 34, 14
```

```
  CTEXT "Target OS", 1, 66, 14, 60, 14
  LISTBOX IDD_LB1, 66, 30, 60, 44, LBS_NOTIFY |
             WS_BORDER | WS_VSCROLL | WS_TABSTOP
}

PropSheetDB2 DIALOGEX 0, 0, PROP_SM_CXDLG, PROP_SM_CYDLG
CAPTION "Choose Options"
{
  DEFPUSHBUTTON "All", IDD_ALL, 11, 10, 34, 14
  AUTOCHECKBOX "Check for stack overflows.", IDD_CB1,
               56, 10, 110, 10
  AUTOCHECKBOX "Check array boundaries.", IDD_CB2,
               56, 30, 110, 10
  AUTOCHECKBOX "Prevent assignment to null.", IDD_CB3,
               56, 50, 110, 10
}

PropSheetDB3 DIALOGEX 0, 0, PROP_SM_CXDLG, PROP_SM_CYDLG
CAPTION "Choose Optimization"
{
  DEFPUSHBUTTON "Fastest", IDD_FASTEST, 11, 10, 34, 14
  PUSHBUTTON "Smallest", IDD_SMALLEST, 11, 34, 34, 14
  AUTORADIOBUTTON "Optimize for speed.", IDD_RB1,
                  56, 10, 100, 10
  AUTORADIOBUTTON "Balance speed and size.", IDD_RB2,
                  56, 30, 100, 10
  AUTORADIOBUTTON "Optimize for size.", IDD_RB3,
                  56, 50, 100, 10
}
```

머리부파일 PROP.H의 내용을 아래에 주었다.

```
#define IDM_DIALOG          100
```

```
#define IDM_EXIT            101
#define IDM_HELP            102

#define IDD_OPTIONS         201
#define IDD_OPTIMIZE        202
#define IDD_FASTEST         204
#define IDD_SMALLEST        205
#define IDD_ALL             207

#define IDD_LB1             301

#define IDD_EB1             401

#define IDD_CB1             501
#define IDD_CB2             502
#define IDD_CB3             503

#define IDD_RB1             601
#define IDD_RB2             602
#define IDD_RB3             603

#define IDD_STEXT1          700
```

프로그람의 실행결과를 그림 13-2에 주었다.

모든 페지의 대화함수들에서 주의를 돌려야 할 점은 사용자가 설정을 변경하였다면 반드시 PSM_CHANGED 통보문을 보낸다는것이다. 례하면 [Choose Options]대화칸에서 검사칸의 상태를 변경하면 *PropSheet_Changed( )*가 호출된다. *PSM_CHANGED* 통보문을 보내면 [Apply]단추의 상태가 무효로부터 유효로 바뀐다. 일반적으로 속성설정이 변경되면 그것을 특성표에 통지해야 한다. 특성표의 첫 페지의 [Options]단추와 [Optimize]단추를 처리하는 부분에서 프로그람코드를 사용하여 페지선택을 조종하는 방법을 보여 주고 있다. [Options]단추를 누르면 두번째 페지가 선택되며 [Optimize]단추를 누르면 세번째 페지가 선택된다. 이 실례프로그람은 조작상 이 단추를 필요로 하지 않지만(페지의 표쪽을 누르면 그 페지를 선택할수 있기때문에) *PSM_SETCURSEL* 통보문의 사용법을 보여 주기 위하여 단추를 사용하고 있다.

이 실례프로그람의 대화칸은 그 어떤 실제적인 기능도 수행하지 않는 허위적인것으로서 특성표의 사용법을 보여 줄뿐이다. 례하면 사용자가 [OK]단추를 누르든가 다른 페지로 이동하면 PSN_KIILLACTIVE 통보문이 발송되여 사용자가 진행한 변경이 보관되지만 사용자가 [Cancel]단추를 누른 경우에는 현재페지에서 진행된 모든 변경이 무시된다.

[Cancel]단추를 눌러서 다른 페지의 변경내용을 되살릴수는 없다. 이 기능은 자체로 추가해 볼수 있다. PSN_APPLY 통보문을 처리하는 기능도 추가해 보는것이 유익하다.

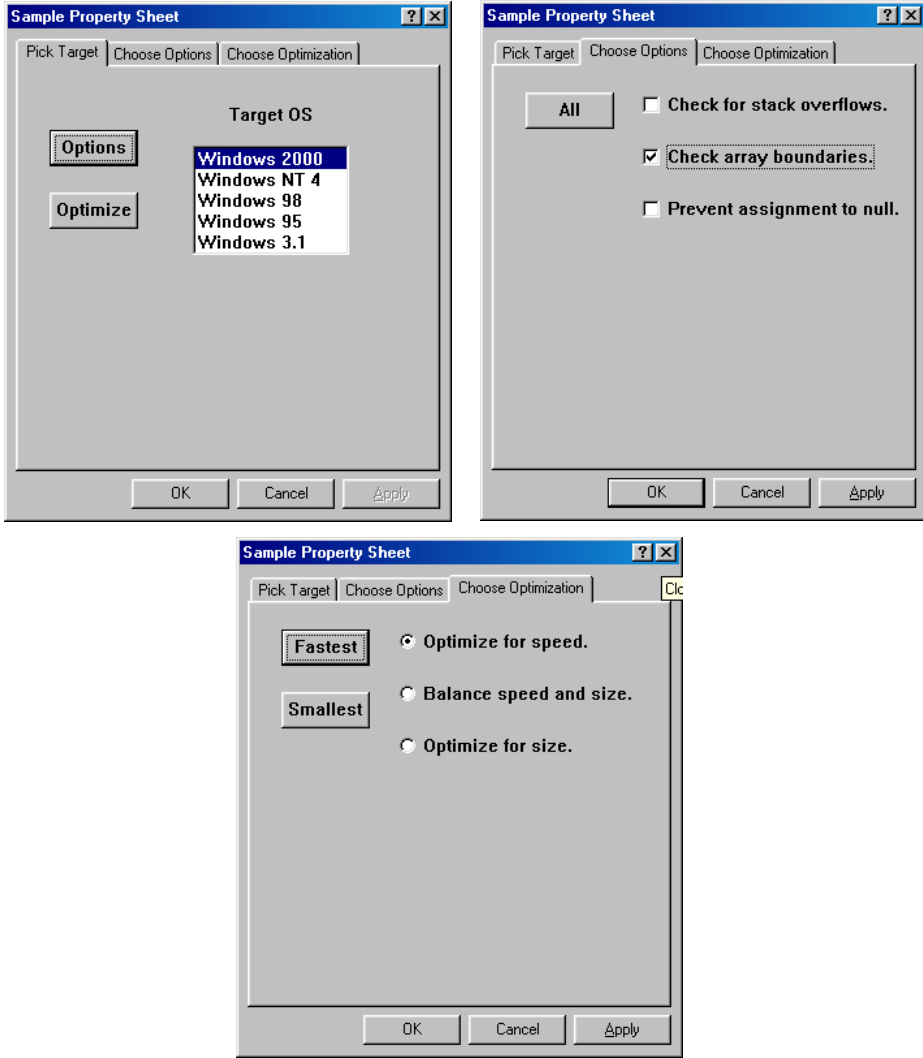



그림 13-2. 특성표의 실례프로그람의 실행결과

다시 한보 전진

PSN_HELP 통보문에 대한 응답

특성표페지를 작성할 때 PSP_HASHELP 기발을 포함시키면 그 페지가 능동으로 되였을 때 특성표에 [Help]단추가 표시된다. 이 단추를 누르면 *PSN_HELP*라는 통지문이 발송된다.

Windows 의 도움말체계를 사용하는 대부분의 응용프로그람들에서는 (도움말

에 대해서는 제 16 장에서 설명한다.) 특성표조종체와 관련된 도움말체계도 제공하고 있다. 그러나 Windows 의 도움말체계는 그것이 세련된것과 마찬가지로 복잡한것이기도 하다. 그러나 특성표에 한정시키면 PSN_HELP 통보문에 대한 응답으로서 간단히 통보칸을 표시하도록 할수 있다.

[Help]단추의 기능을 시험해 보기 위해 특성표실례프로그람을 개조해 보자. 우선 아래에 보여 준것 처럼 PropHdr 구조체의 성원에 *PSH_HASHELP*기발을 추가한다.

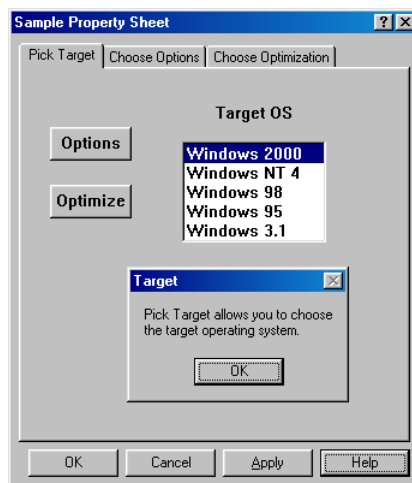
```
PropHdr.dwFlags = PSH_DEFAULT | PSH_HASHELP;
```

다음 특성표의 매 페지에 PSP_HASHELP 기발을 추가한다. 실레로서 첫 페지의 경우를 보여 주었다.

```
PropSheet[0].dwFlags = PSP_DEFAULT | PSP_HASHELP;
```

마지막으로 특성표의 매 페지의 대화함수에서 WM_NOTIFY 통보문을 처리하는 부분에 통지문 PSN_HELP 의 처리를 추가한다.실레로 첫 페지의 대화함수의 경우를 아래에 보여 주었다.

```
case PSN_HELP:
     MessageBox(hdwnd,
                    "Pick Target allows you to choose\n"
                    "the target operating system.",
                    "Target", MB_OK);
     return 1;
```

[Help]단추를 추가한 [Pick Target]페지를 아래에 보여 주었다.

조수의 작성

프로그람작성자의 시점에서 보면 조수는 련속적으로 표시되는 특성표이다. 조수는 이미 설명한 *PROPSHEETPAGE 구조체*와 *PROPSHEETHEADER 구조체*에서 정의되고 *CreatePropertySheetPage( )*함수와 *PropertySheet( )*함수에 의해서 작성된다. 조수를 작성하려면 PROPSHEETHEADER 구조체의 dwFlags 성원에 아래에 보여 주는 기발들 가운데서 하나를 설정해야 한다.

기 발	의 미
PSH_WIZARD97	새로운 형식의 조수를 작성한다.
PSH_WIZARD	낡은 형식의 조수를 작성한다.
PSH_WIZARD_LITE	간소한 형식의 조수를 작성한다.

이 기발들중하나를 포함시키면 매 대화칸이 자동적으로 처음부터 마지막까지 순번대로 표시되게 조수를 작성할수 있다. Windows 2000 에서 사용되는 조수의 형식은 Microsoft 의 Wizard 97 규칙에 준한것이며 형식에 *PSH_WIZARD97* 을 포함시킨다. 이 장에서는 이 형식의 조수에 대해서만 설명한다.

Wizard97 규칙

몇년전에 처음으로 조수가 등장한 때는 조수의 모양이나 조작성을 명백히 정의하는 규칙이 없었다. 이것을 제정하기 위하여 Microsoft 는 *Wizard97* 규칙이라는 새로운 규칙을 제정하였다.

이 규칙은 여러가지 요소들의 크기와 배치, 조수의 시작과 마지막 페지의 모양 및 매 페지의 형식이나 부분제목의 설계 등을 정의하고 있다. 그리 복잡한것은 아니지만 Wizard97 규칙에는 많은 항목들이 있으므로 이 장에서 모두 설명할수는 없다. 그리하여 이 장에서는 조수의 구성방법에 관한 주요항목만을 설명한다. 이미 설명한바와 같이 Windows 2000 에서는 Wizard97 형식의 조수가 사용되고 있으므로 Windows 2000 용의 응용프로그람들도 그대로 따라야 한다. Wizard97 형식의 조수는 Internet Explorer 5.0 이상이 설치되여 있는 체계라면 실행할수 있다. 만일 낡은 조작체계와 아래방향호환성이 있는 조수를 작성하고 싶다면 "Windows NT 4 Programming From the Ground Up"(McGraw Hill) 또는 "Windows 98 Programming"을 참고해야 한다.

외부페지와 내부페지

Windows 2000 형식의 조수는 *외부페지*와 *내부페지*라는 두가지 종류의 페지들로 구성

되여 있다. 외부페지란 조수의 시작과 마감으로 되는 페지를 말한다. 이 페지들은 내부페지들을 둘러 싸게 된다. 외부페지에는 [Welcome]페지와 [Finish]페지가 있다. [Welcome]페지는 조수의 개시를 표시하며 [Finish]페지는 조수의 완료를 표시한다. [Welcome]페지의 실례를 그림 13-3 에 주었다.

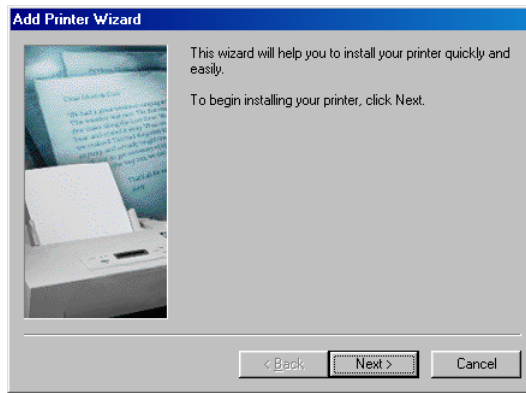



그림 13-3. Windows 2000 의 인쇄기추가 조수의 [Welcome]페지

내부페지는 조수에서 실제로 설정을 진행하기 위한 대화칸으로 된다. 머리부의 제목, 부분제목 및 비트매프도 포함하고 있다. 내부페지의 실례를 그림 13-4 에 주었다.

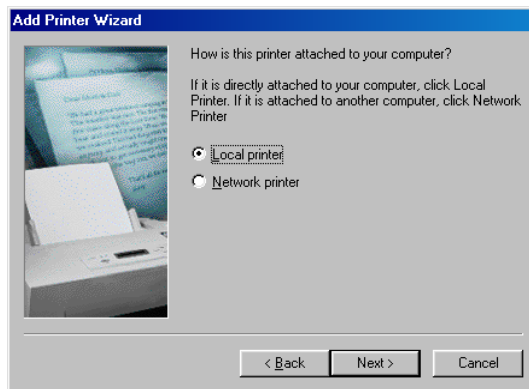



그림 13-4. Windows 2000 의 인쇄기추가 조수의 내부페지

외부페지의 구성

일반적으로 *외부페지*에는 조종체가 들어 있지 않다. 조수의 시작이나 끝을 보여 주는 본문이 표시될뿐이다. [Welcome]페지에는 두드러지게 강조표시된 제목이 있는것이 일반적이다. 례를 들어 만약 로보트의 설정을 진행하는 조수를 작성한다고 하면 그의 [Welcome]페지에 [Welcome to Robot Setting Wizard(로보트의 설정조수를 리용해 주

어서 감사합니다.)]라는 제목을 큼직한 서체로 표시할수 있다.

Wizard97 규칙에 따르면 제목의 서체는 Vernada 의 굵은 체를 사용하며 서체의 크기는 12 포인트로 한다. 그러나 12 포인트는 대화단위이므로 이 값을 화소로 변환하여야 한다.(뒤에서 작성하게 되는 프로그람에서 구체적인 변환방법을 보여 준다.) 제목은 *대화단위*로 (115,8) 위치로부터 표시한다. 시작점을 115 만큼 비우는것은 투명도안(Watermark)을 표시해야 하기때문이다.

[Welcome]페지에 기타 주의사항을 추가하고 나서 마지막에 [To continue,click Next(계속하려면 《다음으로》를 눌러 주십시오)]와 같은 문자렬을 표시한다. 이 본문들은 MS Shell Dlg 서체, 크기는 8 포인트로 한다. 이 서체를 설정하려면 DIALOGEX 명령에 아래의 명령문을 포함시킨다.

```
FONT 8, "MS Shell Dlg"
```

제목이외의 본문은 대화단위로(115,40)의 위치로부터 표시한다.

[Welcome]페지와 [Finish]페지로 되는 대화칸의 크기는 317 × 193 으로 한다.

내부페지의 구성

*내부페지*에는 제목, 부분제목 및 비트매프가 들어 있는 머리부령역이 있다. 이것들은 *PSH_WIZARD97* 형식이 설정되면 자동적으로 확보된다. 내부페지로 되는 대화칸의 크기는 317 × 143 으로 하여야 한다. 이 대화칸에 여러가지 설정을 진행하는 조종체들을 배치한다.

대부분의 경우에 내부페지는 웃부분에 대화단위로 1 만한 너비의 령역을 비우고 측면에 대화단위로 12 만한 너비의 령역을 비워 둔다. 수평방향의 령역을 크게 하고 싶다면 이 값을 7 로 줄일수도 있다.

매 내부페지의 제목과 부분제목은 *PROPSHEETPAGE 구조체*의 pszHeaderTitle 및 pszHeaderSubTitle 에 설정한다. 이 성원들을 유효하게 하려면 형식에 PSP_USEHEADERTITLE 기발과 PSP_USEHEADERSUBTITLE 기발을 포함시켜야 한다.

투명도안과 머리부비트매프

조수의 [Welcome]페지와 [Finish]페지의 왼쪽에는 큰 비트매프가 표시된다. 내부페지에서는 이것이 표시되지 않는다. 이 비트매프는 조수의 투명도안으로서의 역할을 수행하며 투명도안(Watermark)이라고 부른다.

기술적으로는 특별한 의미가 없는것이지만 Wizard97 규칙에서는 *투명도안*을 리용할것을 강하게 주장하고 있다. 투명도안의 크기는 화소단위로 164 × 314 로 한다. 투명도안으로 리용될 비트매프는 *PROPSHEETHEADER 구조체*의 pszbmWatermark 성원에 설정한다. 뒤에서 작성하게 될 실례프로그람에서 투명도안으로 리용되는 비트매프를 작성하고 그것을 BP1.BMP 라는 파일이름으로 보관해 둔다.

*머리부비트매프*는 내부페지의 머리부에 표시된다. 머리부비트매프의 크기는 화소단위로 49×49로 한다. 이것도 기술적으로는 의미가 없는것이지만 Wizar97규칙에서 강하게 주장되고 있으므로 머리부비트매프를 반드시 리용해야 한다. 머리부비트매프는 PROPSHEETHEADER구조체의 pszbmHeader성원에 설정한다. 뒤에서 작성하는 실례프로그람에서 리용할 머리부비트매프를 작성하고 그것을 BP2.BMP 라는 파일이름으로 보관해 둔다.

조수의 단추를 유효상태로 하기

PSH_WIZARD97기발을 포함시키면 특성표가 자동적으로 조수형식으로 표시되게 된다. 하지만 조수가 원만하게 동작하게 하려면 몇가지 기능을 더 추가하여야 한다.

우선 프로그람코드에서 단추의 상태를 유효 또는 무효로 설정하여야 한다. 례를 들어 [Welcome]페지에서는 [Next]단추를 유효로 하고 [Back]단추를 무효로 한다. [Finish]페지에서는 [Back]단추와 [Finish]단추를 유효로 한다.

내부페지에서는 [Next]와 [Back]를 유효로 하여야 한다. 그러자면 *PSM_SETWIZBUTTONS*통보문을 보내든가 또는 PropSheet_SetWizButtons( )마크로를 사용한다. 이미 설명한것처럼 *PropSheet_SetWizButtons( )*의 선언은 아래와 같이 되여있다.

VOID PropSheet_SetWizButtons(hPropSheet, Flags);

hPropSheet 는 특성표조종체의 손잡이이다. Flags에는 유효로 하려는 단추의 종류를 설정한다. 설정한 단추만이 유효로 되며 다른 단추는 무효로 된다. 단추의 종류를 가리키는 마크로들을 아래에 보여 주었다.

PSWIZB_BACK　　　　PSWIZB_NEXT
PSWIZB_FINISH　　　　PSWIZB_DISABLEDFINISH

PSWIZB_DISABLEDFINISH 는 무효상태의 [Finish]단추를 작성한다. 사용자가 반드시 입력해야 하는 항목을 설정하지 않은 경우에는 [Finish]단추를 무효로 한다. 이 마크로들을 두개이상 OR 연산자로 조합할수도 있다. 실례로 [Back]단추와 [Finish]단추를 유효로 한다면 아래와 같이 설정한다.

```
PropSheet_SetWizButtons(PSWIZB_BACK | PSWIZB_FINISH);
```

조수의 실례프로그람

실례 13-2 의 프로그람은 조수를 작성하는 실례프로그람이다. 이 프로그람은 앞에서

설명한 특성표 실례프로그람을 조수로 개조한것이다.

실례 13-2. Wiz 프로그람

```
// 조수의 실례프로그람

#include <windows.h>
#include <cstring>
#include <cstdio>

#include <commctrl.h>
#include "wiz.h"

#define NUMSTRINGS 5
#define NUMPAGES 5

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK DialogFunc0(HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK DialogFunc1(HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK DialogFunc2(HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK DialogFuncWel(HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK DialogFuncComp(HWND, UINT, WPARAM, LPARAM);

char szWinName[] = "MyWin"; // 창문클라스의 이름

HINSTANCE hInst;

HWND hDlg; // 대화칸의 손잡이
HPROPSHEETPAGE hPs[NUMPAGES];
HWND hPropSheet;
HWND hPage[NUMPAGES];

char list[][40] = {
  "Windows 2000",
  "Windows NT 4",
  "Windows 98",
  "Windows 95",
  "Windows 3.1"
};

int cb1=0, cb2=0, cb3=0;
int rb1=1, rb2=0, rb3=0;
```

```
int lb1sel=0;

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;
  HACCEL hAccel;
  INITCOMMONCONTROLSEX cc;

  // 창문클라스의 정의
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst;      // 실체의 손잡이
  wcl.lpszClassName = szWinName; // 창문클라스의 이름
  wcl.lpfnWndProc = WindowFunc;  // 창문함수
  wcl.style = 0;                 // 체계설정의 형식

  wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
  wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
  wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

  wcl.lpszMenuName = "PropSheetMenu"; // 기본차림표

  wcl.cbClsExtra = 0; // 보조기억기령역은 필요 없다.
  wcl.cbWndExtra = 0;

  // 창문의 배경색을 흰색으로 한다.
  wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

  // 창문클라스를 등록한다.
  if(!RegisterClassEx(&wcl)) return 0;

  /* 창문클라스가 등록되였으므로
     창문을 작성할수 있다. */
  hwnd = CreateWindow(
    szWinName, // 창문클라스의 이름
    "Demonstrate a Wizard", // 제목
    WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
    CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
    CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
```

```
    CW_USEDEFAULT, // 너비는 Windows가 결정하게 한다.
    CW_USEDEFAULT, // 높이는 Windows가 결정하게 한다.
    NULL,          // 어미창문은 없다.
    NULL,          // 차림표는 없다.
    hThisInst,     // 실체의 손잡이
    NULL           // 추가파라메터는 없다.
  );

  hInst = hThisInst; // 현재의 실체손잡이를 보관한다.

  // 건반가속기를 적재한다.
  hAccel = LoadAccelerators(hThisInst, "PropSheetMenu");

  // 공통조종체를 초기화한다.
  cc.dwSize = sizeof(INITCOMMONCONTROLSEX);
  cc.dwICC = ICC_TAB_CLASSES;
  InitCommonControlsEx(&cc);

  // 창문을 표시한다.
  ShowWindow(hwnd, nWinMode);
  UpdateWindow(hwnd);

  // 통보문순환고리를 작성한다.
  while(GetMessage(&msg, NULL, 0, 0))
  {
    if(!TranslateAccelerator(hwnd, hAccel, &msg)) {
      TranslateMessage(&msg); // 건반통보를 변환한다.
      DispatchMessage(&msg); // Windows 2000에 조종을 넘긴다.
    }
  }

  return msg.wParam;
}

/* 이 함수는 Windows 2000으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받아 들인다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  int response;
```

```
PROPSHEETPAGE PropSheet[NUMPAGES];
PROPSHEETHEADER PropHdr;

switch(message) {
  case WM_CREATE:
    break;
  case WM_COMMAND:
    switch(LOWORD(wParam)) {
      case IDM_DIALOG:
        PropSheet[0].dwSize = sizeof(PROPSHEETPAGE);
        PropSheet[0].dwFlags = PSP_DEFAULT | PSP_HIDEHEADER;
        PropSheet[0].hInstance = hInst;
        PropSheet[0].pszTemplate = "WelcomeDB";
        PropSheet[0].pszIcon = NULL;
        PropSheet[0].pfnDlgProc = (DLGPROC) DialogFuncWel;
        PropSheet[0].pszTitle = "";
        PropSheet[0].lParam = 0;
        PropSheet[0].pfnCallback = NULL;

        PropSheet[1].dwSize = sizeof(PROPSHEETPAGE);
        PropSheet[1].dwFlags = PSP_DEFAULT |
                               PSP_USEHEADERTITLE |
                               PSP_USEHEADERSUBTITLE;
        PropSheet[1].hInstance = hInst;
        PropSheet[1].pszTemplate = "PropSheetDB1";
        PropSheet[1].pszIcon = NULL;
        PropSheet[1].pfnDlgProc = (DLGPROC) DialogFunc0;
        PropSheet[1].pszTitle = "";
        PropSheet[1].lParam = 0;
        PropSheet[1].pfnCallback = NULL;
        PropSheet[1].pszHeaderTitle = "Pick Target";
        PropSheet[1].pszHeaderSubTitle = "Choose a target operating system.";

        PropSheet[2].dwSize = sizeof(PROPSHEETPAGE);
        PropSheet[2].dwFlags = PSP_DEFAULT  |
                               PSP_USEHEADERTITLE |
                               PSP_USEHEADERSUBTITLE;;
        PropSheet[2].hInstance = hInst;
        PropSheet[2].pszTemplate = "PropSheetDB2";
```

```
PropSheet[2].pszIcon = NULL;
PropSheet[2].pfnDlgProc = (DLGPROC) DialogFunc1;
PropSheet[2].pszTitle = "";
PropSheet[2].lParam = 0;
PropSheet[2].pfnCallback = NULL;
PropSheet[2].pszHeaderTitle = "Choose Options";
PropSheet[2].pszHeaderSubTitle = "Select compilation options.";

PropSheet[3].dwSize = sizeof(PROPSHEETPAGE);
PropSheet[3].dwFlags = PSP_DEFAULT  |
                       PSP_USEHEADERTITLE |
                       PSP_USEHEADERSUBTITLE;
PropSheet[3].hInstance = hInst;
PropSheet[3].pszTemplate = "PropSheetDB3";
PropSheet[3].pszIcon = NULL;
PropSheet[3].pfnDlgProc = (DLGPROC) DialogFunc2;
PropSheet[3].pszTitle = "";
PropSheet[3].lParam = 0;
PropSheet[3].pfnCallback = NULL;
PropSheet[3].pszHeaderTitle = "Choose Optimizations";
PropSheet[3].pszHeaderSubTitle = "Determine how to optimize.";

PropSheet[4].dwSize = sizeof(PROPSHEETPAGE);
PropSheet[4].dwFlags = PSP_DEFAULT | PSP_HIDEHEADER;;
PropSheet[4].hInstance = hInst;
PropSheet[4].pszTemplate = "CompletionDB";
PropSheet[4].pszIcon = NULL;
PropSheet[4].pfnDlgProc = (DLGPROC) DialogFuncComp;
PropSheet[4].pszTitle = "";
PropSheet[4].lParam = 0;
PropSheet[4].pfnCallback = NULL;

hPs[0] = CreatePropertySheetPage(&PropSheet[0]);
hPs[1] = CreatePropertySheetPage(&PropSheet[1]);
hPs[2] = CreatePropertySheetPage(&PropSheet[2]);
hPs[3] = CreatePropertySheetPage(&PropSheet[3]);
hPs[4] = CreatePropertySheetPage(&PropSheet[4]);

PropHdr.dwSize = sizeof(PROPSHEETHEADER);
PropHdr.dwFlags = PSH_WIZARD97 | // 조수로 한다.
```

```
                                    PSH_HEADER |
                                    PSH_WATERMARK;
            PropHdr.hwndParent = hwnd;
            PropHdr.hInstance = hInst;
            PropHdr.pszIcon = NULL;
            PropHdr.pszCaption = "";
            PropHdr.nPages = NUMPAGES;
            PropHdr.nStartPage = 0;
            PropHdr.phpage = hPs;
            PropHdr.pfnCallback = NULL;
            PropHdr.pszbmWatermark = "wizbmp1";
            PropHdr.pszbmHeader = "wizbmp2";

            PropertySheet(&PropHdr);
            break;
          case IDM_EXIT:
            response = MessageBox(hwnd, "Quit the Program?",
                                  "Exit", MB_YESNO);
            if(response == IDYES) PostQuitMessage(0);
            break;
          case IDM_HELP:
            MessageBox(hwnd, "Try the Wizard", "Help", MB_OK);
            break;
        }
        break;
      case WM_DESTROY: // 프로그람을 끝낸다.
        PostQuitMessage(0);
        break;
      default:
        /* 이 switch 문에서 지정된것 이외의 통보문은
           Windows 2000 에 처리를 맡긴다. */
        return DefWindowProc(hwnd, message, wParam, lParam);
    }
    return 0;
  }

  // 첫번째 대화함수
  BOOL CALLBACK DialogFunc0(HWND hdwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
```

```
{
  static long index;
  int i;
  char str[80];

  switch(message) {
    case WM_NOTIFY:
      switch(((NMHDR *) lParam)->code) {
        case PSN_SETACTIVE: // 페지가 입력초점을 얻었다.
          hPropSheet = ((NMHDR *) lParam)->hwndFrom;
          index = lb1sel;
          PropSheet_SetWizButtons(hPropSheet, PSWIZB_NEXT | PSWIZB_BACK);
          SetWindowLong(hdwnd, DWL_MSGRESULT, 0);
          return 1;
        case PSN_KILLACTIVE: // 페지가 입력초점을 잃었다.
          lb1sel = index;
          SetWindowLong(hdwnd, DWL_MSGRESULT, 0);
          return 1;
      }
      break;
    case WM_COMMAND:
      switch(LOWORD(wParam)) {
       case IDD_LB1: // 목록칸의 LBN_DBLCLK 를 처리한다.
          PropSheet_Changed(hPropSheet, hdwnd);
          // 사용자가 선택을 진행하였는가를 확인한다.
          if(HIWORD(wParam)==LBN_DBLCLK) {
            index = SendDlgItemMessage(hdwnd, IDD_LB1,
                    LB_GETCURSEL, 0, 0);  // 색인을 얻는다.
            sprintf(str, "%s", list[index]);

            MessageBox(hdwnd, str, "Selection Made", MB_OK);
          }
          return 1;
      }
      break;
    case WM_INITDIALOG: // 목록칸을 초기화한다.
      for(i=0; i<NUMSTRINGS; i++)
        SendDlgItemMessage(hdwnd, IDD_LB1,
                   LB_ADDSTRING, 0, (LPARAM)list[i]);
```

```
        // 첫 항목을 선택한다.
        SendDlgItemMessage(hdwnd, IDD_LB1, LB_SETCURSEL, lb1sel, 0);

        return 1;
    }

    return 0;
}

// 두번째 대화함수
BOOL CALLBACK DialogFunc1(HWND hdwnd, UINT message,
                          WPARAM wParam, LPARAM lParam)
{
    switch(message) {
      case WM_NOTIFY:
        switch(((NMHDR *) lParam)->code) {
          case PSN_SETACTIVE:// 페지가 입력초점을 얻었다.
            hPropSheet = ((NMHDR *) lParam)->hwndFrom;
            PropSheet_SetWizButtons(hPropSheet,
                          PSWIZB_NEXT | PSWIZB_BACK);
            SetWindowLong(hdwnd, DWL_MSGRESULT, 0);
            return 1;
          case PSN_KILLACTIVE: // 페지가 입력초점을 잃었다.
            cb1 = SendDlgItemMessage(hdwnd, IDD_CB1,
                          BM_GETCHECK, 0, 0);
            cb2 = SendDlgItemMessage(hdwnd, IDD_CB2,
                          BM_GETCHECK, 0, 0);
            cb3 = SendDlgItemMessage(hdwnd, IDD_CB3,
                          BM_GETCHECK, 0, 0);
            SetWindowLong(hdwnd, DWL_MSGRESULT, 0);
          return 1;
        }
        break;
      case WM_COMMAND:
        switch(LOWORD(wParam)) {
          case IDD_CB1:
          case IDD_CB2:
          case IDD_CB3:
            PropSheet_Changed(hPropSheet, hdwnd);
```

```
            return 1;
          case IDD_ALL:
            PropSheet_Changed(hPropSheet, hdwnd);
            SendDlgItemMessage(hdwnd, IDD_CB1, BM_SETCHECK,
                               BST_CHECKED, 0);
            SendDlgItemMessage(hdwnd, IDD_CB2, BM_SETCHECK,
                               BST_CHECKED, 0);
            SendDlgItemMessage(hdwnd, IDD_CB3, BM_SETCHECK,
                               BST_CHECKED, 0);
            return 1;
        }
        break;
      case WM_INITDIALOG: // 검사칸을 초기화한다.
        SendDlgItemMessage(hdwnd, IDD_CB1, BM_SETCHECK, cb1, 0);
        SendDlgItemMessage(hdwnd, IDD_CB2, BM_SETCHECK, cb2, 0);
        SendDlgItemMessage(hdwnd, IDD_CB3, BM_SETCHECK, cb3, 0);
        return 1;
    }

    return 0;
  }

  // 세번째 대화함수
  BOOL CALLBACK DialogFunc2(HWND hdwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
  {
    switch(message) {
      case WM_NOTIFY:
        switch(((NMHDR *) lParam)->code) {
          case PSN_SETACTIVE: // 페지가 입력초점을 얻었다.
            hPropSheet = ((NMHDR *) lParam)->hwndFrom;
            PropSheet_SetWizButtons(hPropSheet,
                              PSWIZB_BACK | PSWIZB_NEXT);
            SetWindowLong(hdwnd, DWL_MSGRESULT, 0);
            return 1;
          case PSN_WIZFINISH: // [Finish]단추가 눌리웠다.
          case PSN_KILLACTIVE: // 페지가 입력초점을 잃었다.
            rb1 = SendDlgItemMessage(hdwnd, IDD_RB1,
                               BM_GETCHECK, 0, 0);
```

```
          rb2 = SendDlgItemMessage(hdwnd, IDD_RB2,
                          BM_GETCHECK, 0, 0);
          rb3 = SendDlgItemMessage(hdwnd, IDD_RB3,
                          BM_GETCHECK, 0, 0);
         SetWindowLong(hdwnd, DWL_MSGRESULT, 0);
         return 1;
       }
       break;
     case WM_COMMAND:
       switch(LOWORD(wParam)) {
         case IDD_RB1:
         case IDD_RB2:
         case IDD_RB3:
           PropSheet_Changed(hPropSheet, hdwnd);
           return 1;
         case IDD_FASTEST:
           PropSheet_Changed(hPropSheet, hdwnd);
           SendDlgItemMessage(hdwnd, IDD_RB2, BM_SETCHECK, 0, 0);
           SendDlgItemMessage(hdwnd, IDD_RB3, BM_SETCHECK, 0, 0);
           SendDlgItemMessage(hdwnd, IDD_RB1, BM_SETCHECK, 1, 0);
           return 1;
         case IDD_SMALLEST:
           PropSheet_Changed(hPropSheet, hdwnd);
           SendDlgItemMessage(hdwnd, IDD_RB1, BM_SETCHECK, 0, 0);
           SendDlgItemMessage(hdwnd, IDD_RB2, BM_SETCHECK, 0, 0);
           SendDlgItemMessage(hdwnd, IDD_RB3, BM_SETCHECK, 1, 0);
           return 1;
       }
       break;
     case WM_INITDIALOG: // 단일선택단추를 초기화한다.
       SendDlgItemMessage(hdwnd, IDD_RB1, BM_SETCHECK, rb1, 0);
       SendDlgItemMessage(hdwnd, IDD_RB2, BM_SETCHECK, rb2, 0);
       SendDlgItemMessage(hdwnd, IDD_RB3, BM_SETCHECK, rb3, 0);
       return 1;
   }
   return 0;
 }
```

```
// [Welcome]대화칸
BOOL CALLBACK DialogFuncWel(HWND hdwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  PAINTSTRUCT ps;
  HDC dc;
  static HFONT hFontTitle;
  static char *welcomeStr = "Welcome to the Wizard";
  RECT r;

  switch(message) {
    case WM_PAINT:
      dc = BeginPaint(hdwnd, &ps);
      r.right = 115; r.bottom = 8;
      MapDialogRect(hdwnd, &r); // 화소단위로 변환한다.
      SelectObject(dc, hFontTitle); // 큰 서체를 선택한다.
      TextOut(dc, r.right, r.bottom, welcomeStr,
              strlen(welcomeStr));
      EndPaint(hdwnd, &ps);
      break;
    case WM_NOTIFY:
      switch(((NMHDR *) lParam)->code) {
        case PSN_SETACTIVE:// 페지가 입력초점을 얻었다.
          r.bottom = 12;
          MapDialogRect(hdwnd, &r); // 화소단위로 변환한다.
          // 조수제목을 위한 서체를 작성한다.
          hFontTitle = CreateFont(r.bottom, 0, 0, 0, FW_MEDIUM,
                         0, 0, 0, ANSI_CHARSET,
                         OUT_DEFAULT_PRECIS,
                         CLIP_DEFAULT_PRECIS,
                         DEFAULT_QUALITY,
                         DEFAULT_PITCH | FF_DONTCARE,
                         "Verdana");

          hPropSheet = ((NMHDR *) lParam)->hwndFrom;
          PropSheet_SetWizButtons(hPropSheet,
                          PSWIZB_NEXT);
          SetWindowLong(hdwnd, DWL_MSGRESULT, 0);
          return 1;
```

```
            case PSN_KILLACTIVE: // 페지가 입력초점을 잃었다.
              SetWindowLong(hdwnd, DWL_MSGRESULT, 0);
              DeleteObject(hFontTitle);
              return 1;
          }
          break;
    }

    return 0;
  }

  // [Finish]대화칸
  BOOL CALLBACK DialogFuncComp(HWND hdwnd, UINT message,
                               WPARAM wParam, LPARAM lParam)
  {
    PAINTSTRUCT ps;
    HDC dc;
    static HFONT hFontTitle;
    static char *compStr = "Completing the Wizard.";
    RECT r;

    switch(message) {
      case WM_PAINT:
        dc = BeginPaint(hdwnd, &ps);
        r.right = 115; r.bottom = 8;
        MapDialogRect(hdwnd, &r); // 화소단위로 변환한다.
        SelectObject(dc, hFontTitle); // 큰 서체를 선택한다.
        TextOut(dc, r.right, r.bottom, compStr,
                strlen(compStr));
        EndPaint(hdwnd, &ps);
        break;
      case WM_NOTIFY:
        switch(((NMHDR *) lParam)->code) {
          case PSN_SETACTIVE:// 페지가 입력초점을 얻었다.
            r.bottom = 12;
            MapDialogRect(hdwnd, &r); // 화소단위로 변환한다.
            // 조수제목을 위한 서체를 작성한다.
            hFontTitle = CreateFont(r.bottom, 0, 0, 0, FW_MEDIUM,
                                    0, 0, 0, ANSI_CHARSET,
```

```
                              OUT_DEFAULT_PRECIS,
                              CLIP_DEFAULT_PRECIS,
                              DEFAULT_QUALITY,
                              DEFAULT_PITCH | FF_DONTCARE,
                              "Verdana");

                hPropSheet = ((NMHDR *) lParam)->hwndFrom;
                PropSheet_SetWizButtons(hPropSheet,
                               PSWIZB_BACK | PSWIZB_FINISH);
                SetWindowLong(hdwnd, DWL_MSGRESULT, 0);
                return 1;
              case PSN_KILLACTIVE: // 페지가 입력초점을 잃었다.
               SetWindowLong(hdwnd, DWL_MSGRESULT, 0);
               DeleteObject(hFontTitle);
               return 1;
           }
           break;
       }

       return 0;
     }
```

이 프로그람은 아래와 같은 자원파일을 필요로 한다. 두개의 비트매프가 추가된 점에 주의를 돌려야 한다. 이미 설명한바와 같이 BP1.BMP 는 투명도안으로 되며 BP2.BMP 는 *머리부비트매프*이다. 프로그람을 번역하기전에 이 비트매프들을 미리 작성하여야 한다.

```
     // 조수의 자원파일
     #include <windows.h>
     #include <commctrl.h>
     #include "wiz.h"
     wizbmp1 BITMAP bp1.bmp
     wizbmp2 BITMAP bp2.bmp
     PropSheetMenu MENU
     {
       POPUP "&Wizard Demo"
       {
         MENUITEM "&Start Wizard\tF2", IDM_DIALOG
```

```
    MENUITEM "E&xit\ tCtrl+X", IDM_EXIT
  }
  MENUITEM "&Help", IDM_HELP
}
PropSheetMenu ACCELERATORS
{
  VK_F2, IDM_DIALOG, VIRTKEY
  VK_F1, IDM_HELP, VIRTKEY
  "^X", IDM_EXIT
}
WelcomeDB DIALOGEX 0, 0, 317, 193
CAPTION "Compilation Options Wizard"
FONT 8, "MS Shell Dlg"
{
  LTEXT "To continue, click Next.", IDD_STEXT2, 115, 40, 100, 14
}
PropSheetDB1 DIALOGEX 0, 0, 317, 143
CAPTION "Compilation Options Wizard"
FONT 8, "MS Shell Dlg"
{
  CTEXT "Choose Target OS", IDD_STEXT1, 21, 1, 60, 14
  LISTBOX IDD_LB1, 21, 24, 60, 52, LBS_NOTIFY |
           WS_BORDER | WS_VSCROLL | WS_TABSTOP
}
PropSheetDB2 DIALOGEX 0, 0, 317, 143
CAPTION "Compilation Options Wizard"
FONT 8, "MS Shell Dlg"
{
  DEFPUSHBUTTON "All", IDD_ALL, 21, 1, 34, 14
  AUTOCHECKBOX "Check for stack overflows.", IDD_CB1,
               21, 31, 92, 10
  AUTOCHECKBOX "Check array boundaries.", IDD_CB2,
               21, 51, 92, 10
  AUTOCHECKBOX "Prevent assignment to null.", IDD_CB3,
```

```
                  21, 71, 100, 10
}
PropSheetDB3 DIALOGEX 0, 0, 317, 143
CAPTION "Compilation Options Wizard"
FONT 8, "MS Shell Dlg"
{
  DEFPUSHBUTTON "Fastest", IDD_FASTEST, 21, 1, 34, 14
  PUSHBUTTON "Smallest", IDD_SMALLEST, 21, 25, 34, 14
  AUTORADIOBUTTON "Optimize for speed.", IDD_RB1,
                  66, 1, 90, 10
  AUTORADIOBUTTON "Balance speed and size.", IDD_RB2,
                  66, 21, 90, 10
  AUTORADIOBUTTON "Optimize for size.", IDD_RB3,
                  66, 41, 90, 10
}
CompletionDB DIALOGEX 0, 0, 317, 193
CAPTION "Compilation Options Wizard"
FONT 8, "MS Shell Dlg"
{
  LTEXT "To close, click Finish.", IDD_STEXT3, 115, 40, 100, 14
}
```

머리부파일 WIZ.H의 내용을 아래에 보여 주었다.

```
#define IDM_DIALOG          100
#define IDM_EXIT            101
#define IDM_HELP            102

#define IDD_OPTIONS         201
#define IDD_OPTIMIZE        202
#define IDD_FASTEST         204
#define IDD_SMALLEST        205
#define IDD_ALL             207

#define IDD_LB1             301
```

```
#define IDD_EB1           401

#define IDD_CB1           501
#define IDD_CB2           502
#define IDD_CB3           503

#define IDD_RB1           601
#define IDD_RB2           602
#define IDD_RB3           603

#define IDD_STEXT1        700
#define IDD_STEXT2        701
#define IDD_STEXT3        702
```

프로그람의 실행결과를 그림 13-5 에 보여 주었다.

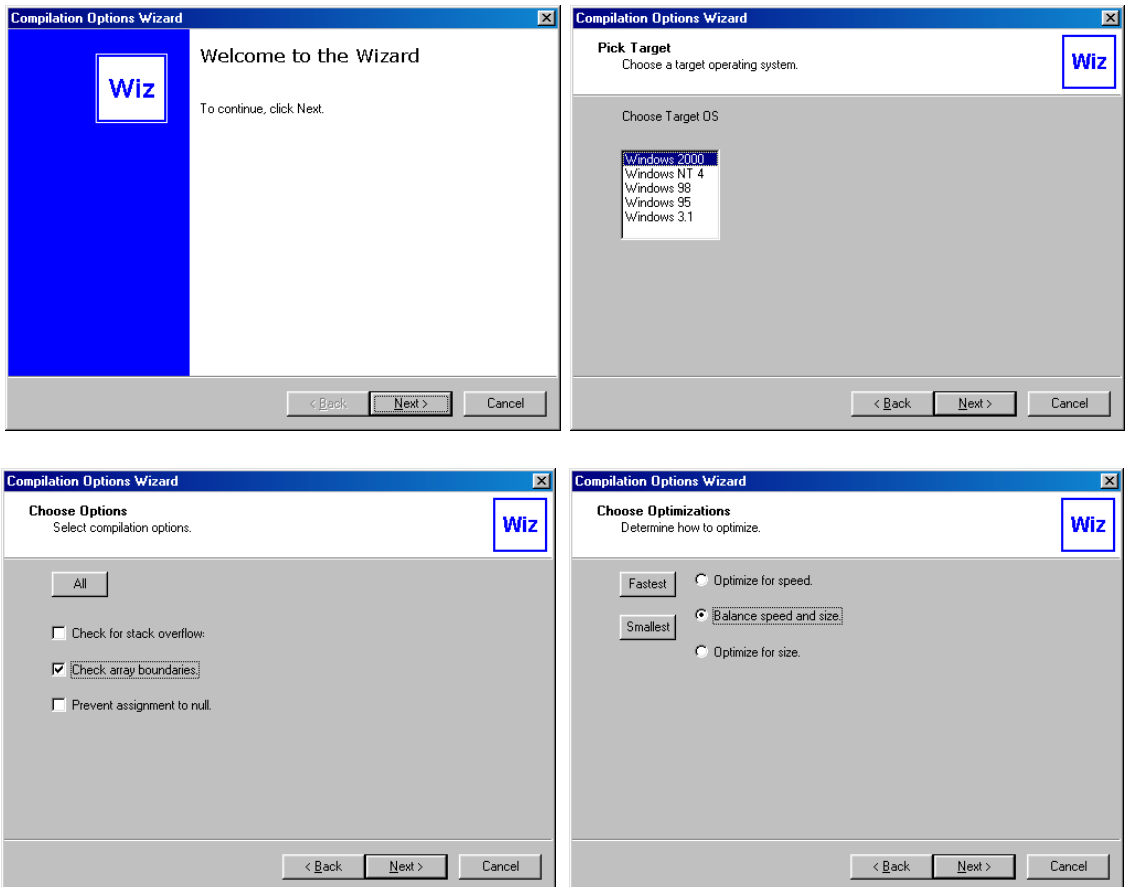



그림 13-5. 조수의 실례프로그람의 실행결과

조수실례프로그람의 상세

조수실례프로그람의 많은 부분은 특성표 실례프로그람과 같다. 그러나 일부 변경된 부분도 있다.

우선 PropHdr구조체의 dwFlags성원에 PSH_WIZARD97기발이 추가된 점에 주목해야 한다. 이미 설명한것처럼 이 기발은 특성표를 최신형식의 조수로 변환한다. PropHdr구조체에는 필요한 기발들과 함께 투명도안과 머리부비트매프도 추가되여 있다.

Wizard97 규칙에 따라 자원파일에 [Welcome]대화칸과 [Finish]대화칸이 추가되여 있고 그것들의 대화함수가 프로그람에 추가되여 있다. 이 프로그람에서는 조수를 실현하는데 필요한 대화칸의 수가 세개가 아니라 다섯개로 되여 있으므로 NUMPAGES 마크로의 수를 5로 변경하였다.

[Welcome]대화칸과 [Finish]대화칸의 대화함수는 각각 DialogFuncWel( ) 및 DialogFuncComp( )이다. 앞에서 본 프로그람의 대화칸에 정의되여 있던 여러가지 조종체들이 내부페지에서 사용되고 있으나 그것들의 위치는 Wizard97 규칙에 맞추어 변경되였다.

이 프로그람에서는 새로운 페지가 능동상태로 되면 적절한 단추들만을 유효로 하고 있다. 필요에 따라 [Back], [Next] 및 [Finish]단추를 유효로 하고 있다. 그것을 실현하기 위한 프로그람코드가 WM_NOTIFY통보문을 처리하는 부분에 추가되여 있다.

조수의 매 페지는 순서대로 표시되여야 하므로 [Pick Target]페지의 [Options]단추와 [Optimize]단추가 삭제되여 있다. 조수를 사용할 때 순서를 무시하여 페지를 능동으로 하는것은 Windows 2000의 정확한 형식이 아니다.

[Welcome]페지와 [Finish]페지에 제목을 표시하기

이미 설명한것처럼 내부페지의 제목은 대화단위로 12포인트, 굵은체, Verdana서체로 한다. 대화단위를 화소로 변환하려면 *MapDialogRect( )*함수를 사용한다. 아래에 선언을 보여 주었다.

```
BOOL MapDialogRect(HWND hDialog, RECT *rect);
```

hDialog에 대화칸의 손잡이를 설정한다. 화소단위로 변환하기전의 크기를 rect구조체에 설정한다. 함수의 호출이 성공한 경우는 령 아닌 값이 돌려 지며 실패한 경우에는 령이 돌려 진다.

대화단위의 12를 화소로 변환하려면 RECT구조체의 bottom성원에 12를 설정한 다음 MapDialogRect( )를 호출한다. 같은 rect 구조체의 bottom 성원에 화소단위로 변환된 크기가 돌려 진다. 이 값은 제목을 위한 서체의 크기를 설정하는데 사용된다.

제목을 위한 서체가 외부페지가 능동으로 될 때마다 작성되고 페지가 바뀔 때마다 파괴되고 있는 점에도 주의를 돌려야 한다. 이러한 처리가 필요한것은 서체가 필요 없게 되면 해제해야 하는 자원이기때문이다.

대화단위로부터 화소에로의 변환은 제목이 표시될 때도 진행되여야 한다. 이미 설명한바와 같이 제목은 대화단위로 (115,8)의 위치에 표시된다. 이 프로그람에서는 다시 한번 MapDialogRect( )를 사용하여 이 위치를 화소로 변환하고 있다.

노력한것만한 가치가 있다

특성표와 조수는 Windows 2000 응용프로그람에 세련된 조작성을 제공해 준다. 이것들을 작성하려면 약간한 품이 들지만 복잡한 입력과정을 인도하여 주므로 노력한것만한 가치가 있는것이다.

특히 조수를 사용하면 한개의 입력화면으로는 표시할수 없는 복잡한 항목들을 설정하는 경우에도 그것을 사용자에게 알기 쉽게 제공할수 있다. 이제는 조수의 작성방법을 정통하였으므로 그것을 복잡한 입력처리가 요구되는 장면들에서 능히 활용할수 있을것이다.

제 14 장

머리부조종체와 월사업표조종체

이 장에서는 머리부조종체와 월사업표조종체라는 두가지 *공통조종체*의 사용방법에 대하여 설명한다.

머리부조종체는 렬머리부들의 띠로 구성되여 있으며 표형식으로 정보를 표시할 때 사용된다. 월사업표조종체는 비교적 새로운 조종체로서 달력을 표시하고 날자를 선택할수 있게 되여 있다. 월사업표조종체는 실물의 달력과 비슷한 모양을 가진다.

머리부조종체

*머리부조종체*는 렬머리부로 구성된 띠이다. Windows 2000 을 사용하고 있다면 반드시 머리부조종체를 본적이 있을것이다. 례를 들어 탐색기에서는 파일의 구체적인 정보를 표시하는 부분에서 머리부조종체가 사용되고 있다. 그러나 머리부조종체는 단순히 정보를 표시하기만 하는것이 아니라 사용자가 매개 렬의 크기를 변경할수 있으며 마우스의 찰칵에 응답할수도 있다. 머리부조종체는 여러가지 장면에서 활용할수 있는 공통조종체이며 정보를 렬형식으로 표시하여 관리하기 위한 표준적인 기능들을 제공해 준다.

머리부조종체의 작성

머리부조종체는 창문클라스에 *WC_HEADER* 를 지정하고 *CreateWindow( )* 혹은 *CreateWindowEx( )*를 호출하여 작성된다. 그때 형식에 WS_CHILD 를 포함시킨다.

일반적으로 머리부조종체를 작성할 때는 너비와 높이를 령으로 해둔다. 그 리유는 프로그람이 실행될 때 머리부조종체의 크기를 어미창문의 의뢰자구역에 맞추어 조정해야 하기때문이다.

또한 현재 선택되여 있는 서체의 크기에도 맞추어야 한다. 다행히 머리부조종체에는 그것이 작성된후에 크기를 변경시키는 기능이 갖추어 져 있으므로 수동적으로 크기를 조정할 필요는 없다.

머리부조종체의 크기는 그것이 작성된후부터 조정하기때문에 대부분의 머리부조종체의 작성은 아래와 같이 CreateWindow( )를 사용하여 진행한다.

```
hHeadWnd = CreateWindow(WC_HEADER, NULL,
                        WS_CHILD | WS_BORDER,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        0, 0, hParent,
                        (HMENU) ID_HEADCONTROL,
                        hInst, NULL);
```

hParent 에는 어미창문의 손잡이를 설정한다. hInst 에는 응용프로그람의 실체손잡이를 설정한다. ID_HEADCONTROL(이 값은 임의로 설정한다.)은 머리부조종체의 ID 이다. 크기를 0 으로 하였으므로 초기에는 머리부조종체가 표시되지 않는다.

체계설정의 머리부조종체는 표식과 경계선만으로 구성되게 된다. 그러나 머리부조종체를 작성할 때에 *HDS_BUTTONS* 형식을 포함시키면 누름단추로 구성된 머리부항목을 작성할수 있다. 이 기능을 사용한 실례프로그람은 이 장의 뒤부분에서 소개한다.

작성직후의 머리부조종체는 비여 있는 상태로 되여 있다. 매 머리부항목의 내용은 각각 추가하여야 한다. 조종체의 크기를 어미창문의 크기에 맞추는것도 필요하게 된다.
이러한 리유로부터 작성 직후의 머리부조종체는 비표시상태로 해 두어야 하는것이다. 머리부항목의 설정과 크기의 조정이 끝난 다음에 머리부조종체를 표시한다. 그러므로 머리부조종체를 사용하기 위한 절차는 다음과 같이 된다.

- 머리부조종체의 적절한 크기를 결정하고 그 크기를 설정한다.
- 머리부조종체의 매 머리부항목을 추가한다.
- 머리부조종체를 표시한다.

이러한 절차를 실행하려면 뒤에서 설명하는 여러가지 통보문들을 머리부조종체에 보내야 한다.

머리부조종체에 통보문을 보내기

머리부조종체는 여러가지 통보문에 응답한다. 흔히 사용되는 통보문들을 표 14-1 에 보여 주었다. 머리부조종체에 통보문을 보내려면 통보문을 받는 측으로 머리부조종체의 손잡이를 지정하여 SendMessage( )를 호출한다. 통보문을 보내는 처리를 쉽게 하기 위한 마크로들도 제공되고 있다. 표 14-1 에 보여 준 통보문들에 대응하는 마크로를 아래에 보여 주었다.

```
BOOL Header_DeleteItem(HWND hHeadWnd, int index);

BOOL Header_GetItem(HWND hHeadWnd, int index,
                        HDITEM *HdItemPtr);

int Header_GetItemCount(HWND hHeadWnd);

int Header_InsertItem(HWND hHeadWnd,int index,
                        HDITEM *HdItemPtr);

BOOL Header_Layout(HWND hHeadWnd, HDLAYOUT *LayoutPtr);

BOOL Header_SetItem(HWND hHeadWnd,int index,
                        HDITEM *HdItemPtr);
```

모든 마크로에 있어서 hHeadWnd에는 머리부조종체의 손잡이를 설정하고 index에는 처리대상으로 되는 항목의 색인을 설정한다. HdItemPtr 는 *HDITEM* 구조체의 지시자

이다. LayoutPtr 는 *HDLAYOUT* 구조체의 지시자이다. 이 구조체들은 머리부조종체의 작성이나 조작에 있어서 특히 중요한것들이므로 상세하게 설명한다.

표 14-1. 머리부조종체의 주요한 통보문

통 보 문	의 미
HDM_DELETEITEM	머리부항목을 삭제한다. 호출이 성공한 경우는 령 아닌 값을 돌려 주며 실패한 경우 령을 돌려 준다. wParam 에 머리부항목의 색인을 설정한다. lParam 에는 령을 설정한다.
HDM_GETITEM	머리부항목의 정보를 얻는다. 호출이 성공한 경우는 령 아닌 값을 돌려 주며 실패한 경우 령을 돌려 준다. wParam 에는 머리부항목의 색인을 설정한다. lParam 에는 얻은 정보를 보관하기 위한 HDITEM 구조체의 지시자를 설정한다. HDITEM 구조체의 성원 mask 의 값은 얻은 정보를 식별하는 것으로 된다.
HDM_GETITEMCOUNT	머리부항목의 수를 돌려 준다. 호출이 성공한 경우는 -1 을 돌려 준다. wParam 과 lParam 에는 령을 설정한다.
HDM_HITTEST	점의 자리표를 보내면 그 위치의 머리부항목의 색인이 돌려 진다. 지정된 점이 머리부항목의 내부에 있지 않는 경우는 -1 이 돌려 진다. wParam 에는 령을 설정한다. lParam 에는 점을 지정하는 HDHITTESTINFO 구조체의 지시자를 설정한다.
HDM_INSERTITEM	머리부항목을 삽입한다. 호출이 성공한 경우는 삽입된 머리부항목의 색인이 돌려 지고 실패한 경우는 -1 이 돌려 진다. wParam 에는 그후에 새로운 머리부항목을 삽입할 색인을 설정한다. 선두에 머리부항목을 삽입하는 경우는 색인에 령을 설정한다. 끝에 머리부항목을 삽입하는 경우는 현재의 머리부조종체에 존재하는 머리부항목의 수보다 큰 값을 색인으로 설정한다. lParam 에는 새로 삽입할 머리부항목의 정보를 포함한 HDITEM 구조체의 지시자를 설정한다.
HDM_LAYOUT	어미창문의 의뢰자구역의 크기를 보내면 그에 맞는 적절한 머리부조종체의 크기가 얻어 진다. 호출이 성공한 경우에는 령 아닌 값이 돌려 지며 실패

	한 경우는 령이 돌려 진다. wParam 에는 령을 설정한다. lParam 에는 HDLAYOUT 구조체의 지시자를 설정한다. HDLAYOUT 구조체의 prc 성원에 어미창문의 의뢰자구역의 크기를 설정한다. 돌림값으로서 HDLAYOUT 구조체의 pwpos 성원에 머리부조종체의 적절한 크기가 돌려 진다.
HDM_SETITEM	머리부항목에 정보를 설정한다. 호출이 성공한 경우는 령 아닌 값이 돌려 지며 실패한 경우에는 령이 돌려 진다. wParam 에는 머리부항목의 색인을 설정한다. lParam 에는 머리부항목의 정보를 보관한 HDITEM 구조체의 지시자를 설정한다. HDITEM 구조체의 성원 mask 에는 설정하는 정보를 식별하는 값을 설정한다.

HDITEM 구조체
머리부조종체의 매 머리부항목은 아래에 보여 준 *HDITEM* 구조체에서 정의된다.

```
typedef struct _HDITEM
{
     UINT mask;
     int cxy;
     LPSTR pszText;
     HBITMAP hbm;
     int cchTextMax;
     int fmt;
     LPARAM lParam;
     int iImage;
     int iOrder;
     UINT type; // Windows 2000 에 새롭게 추가된것
     LPVOID pvFilter; // Windows 2000 에 새롭게 추가된것
}    HDITEM;
```

mask 의 값은 HDITEM 구조체의 어느 성원에 정보가 들어 있는가를 표시한다. 이것은 아래에 보여 준 값들의 조합으로 된다.

값	의　　미
HDI_BITMAP	hbm 에 비트매프의 손잡이가 보관되여 있다.
HDI_FILTER	type 및 pvFilter 에 정보가 보관되여 있다.(Windows 2000 에 새롭게 추가된것)
HDI_FORMAT	fmt 에 양식기발이 보관되여 있다.
HDI_HEIGHT	cxy 에 머리부항목의 높이가 보관되여 있다.
HDI_WIDTH	cxy 에 머리부항목의 너비가 보관되여 있다.
HDI_LPARAM	lParam 에는 정보가 보관되여 있다.
HDI_TEXT	pszText 및 cchTextMax 에 정보가 보관되여 있다.
HDI_IMAGE	iImage 에 정보가 보관되여 있다.
HDI_ORDER	iOrder 에 정보가 보관되여 있다.

cxy 에는 mask 의 값이 HDI_WIDTH 와 HDI_HEIGHT 가운데 어느것인가에 따라 머리부항목의 너비 또는 높이를 설정한다.

pszText 에는 렬의 표식으로 되는 문자렬의 지시자를 설정한다. 이 문자렬의 길이는 cchTextMax 에 설정한다.

머리부항목에 비트매프를 표시하는 경우에는 hbm 에 비트매프의 손잡이를 설정한다. fmt 에 설정되는 값은 머리부항목을 표시할 때의 형식을 가리킨다. 이것은 아래에서 보여 주는 값과 배치를 나타내는 값의 조합으로 된다.

값	의　　미
HDF_STRING	머리부항목에 문자렬을 표시한다.
HDF_BITMAP	머리부항목에 비트매프를 표시한다.
HDF_BITMAP_ON_RIGHT	비트매프를 본문의 오른쪽에 표시한다.
HDF_OWNERDRAW	소유자그리기로 한다.

배치를 가리키는 값을 아래에 보여 주었다.

값	의　　미
HDF_LEFT	왼쪽맞추기
HDF_CENTER	가운데맞추기
HDF_RIGHT	오른쪽맞추기

lParam 에는 응용프로그람자체의 자료를 설정할수 있다.

머리부항목에 화상목록을 포함시키는 경우는 화상의 색인을 iImage 에 설정한다. 그러나 이 장의 실례프로그람에서는 화상목록을 사용하지 않는다.

머리부항목의 정보를 얻을 때는 iOrder 에 머리부항목의 위치를(0 을 시작점으로 하

여 왼쪽에서 오른쪽의 순서로) 설정한다.

type와 pvFilter는 Windows 2000에서 새롭게 추가된것이다. 이 성원들은 자료의 선별기를 작성하는데 사용된다. 그러나 이 장의 실례프로그람에서는 사용되지 않는다.

이식과 관련한 요점 : HDITEM구조체를 이전에는 HD_ITEM구조체라고 불렀다.

HDLAYOUT구조체

HDLAYOUT 구조체는 머리부조종체의 적절한 크기를 얻기 위해 *HDM_LAYOUT* 통보문과 함께 사용된다. 이 경우는 머리부조종체의 어미창문의 의뢰자구역의 직4각형 크기를 주는것이 일반적이다. HDLAYOUT구조체의 정의를 아래에 보여 주었다.

```
typedef struct _HDLAYOUT
{
     RECT *prc;
     WINDOWPOS *pwpos;
}    HDLAYOUT;
```

prc에 머리부조종체가 표시되는 령역의 크기를 가리키는 RECT구조체의 지시자를 설정한다. 머리부조종체가 크기를 계산하고 지정된 령역에 맞는 적절한 크기를 결정한다. 계산된 크기는 pwpos가 가리키는 WINDOWSPOS 구조체의 지시자에 돌려 진다. RECT 구조체에 대해서는 이미 알고 있다. *WINDOWPOS*구조체의 정의를 아래에 보여 주었다.

```
typedef struct tagWINDOWPOS {
     HWND hwnd;
     HWND hwndInsertAfter;
     int x;
     int y;
     int cx;
     int cy;
     UINT flags;
}    WINDOWPOS;
```

hwnd는 머리부조종체의 손잡이이다. hwndInsertAfter는 *Z차례*(화면에서 창문들이 놓이는 순서)에서 하나밑에 있는 창문의 손잡이이다. 창문의 왼쪽 웃모서리의 자리표가 x와 y에 보관된다. cx에 너비가 보관되고 cy에 높이가 보관된다. flags의 값은 창문과 관련되는 여러가지 속성들을 지시한다. 이 장에서는 WINDOWPOS 구조체를 머리

부조종체에 초기화된 그대로의 상태로 사용한다. 구조체의 내용을 변경할 필요는 없다.

이식과 관련한 요점 : HDLAYOUT 구조체를 전에는 HD_LAYOUT 라고 불렀다.

머리부의 크기변경

초기화가 끝난 머리부조종체의 크기를 조정하려면 HDM_LAYOUT 통보문을 보낸다.(또는 Header_Layout( )마크로를 사용한다.) 이 통보문에 대한 응답으로서 돌려 진 크기나 위치를 리용하여 머리부조종체의 크기를 조정한다. 다음에 실례를 보여 주었다.

```
RECT rect;
HDLAYOUT layout;
WINDOWPOS winpos;

// 어미창문의 크기를 얻는다.
GetClientRect(hParent, &rect);

// 의뢰자구역에 맞춘 머리부조종체의 크기를 얻는다.
layout.pwpos = &winpos;
layout.prc = &rect;
Header_Layout(hHeadWnd, &layout);

// 의뢰자구역에 맞추어 머리부조종체의 크기를 변경한다.
MoveWindow(hHeadWnd, winpos.x, winpos.y,
winpos.cx, winpos.cy, 0);
```

우선 GetClientRect( )를 호출하여 머리부조종체를 표시하는 창문의 크기를 얻는다. 다음 이 크기를 Header_Layout( )를 사용하여 머리부조종체에 통지한다. 돌림값으로서 pwpos 성원에 머리부조종체의 적절한 크기와 위치가 돌려 진다. 이 값들을 파라메터로 주어 *MoveWindow( )*를 호출하여 머리부조종체의 크기와 위치를 설정한다.([**다시 한보전진**]을 참고할것)

이미 설명한것처럼 이 시점에서는 아직 머리부조종체가 표시되지 않는다. 머리부항목을 삽입하고 나서 머리부조종체를 표시한다.

다시 한보 전진

창문의 이동

창문의 위치나 크기를 변경시키기 위하여 MoveWindow( )라는 API 함수를 사용한다. 아래에 선언을 보여 주었다.

```
BOOL MoveWindow(HWND hwnd, int NewX, int NewY,
                int NewWidth, int NewHeight, BOOL Repaint);
```

hwnd에 처리대상으로 되는 창문의 손잡이를 설정한다. 새로 이동할 왼쪽웃모서리의 자리표를 NewX와 NewY에 설정한다. 새로운 너비와 높이를 NewWidth와 NewHeight에 설정한다. Repaint에 령 아닌 값을 설정하면 위치나 크기를 변경한후에 창문이 곧 다시그리기된다. 그렇지 않은 경우에는 다시그리기요구가 창문의 통보문렬에 보관되고 후에 처리된다. *MoveWindow( )*는 호출이 성공하면 령 아닌 값을 돌려 주고 실패하면 령을 돌려 준다.

MoveWindow( )를 사용하면 창문의 크기와 위치를 변경시킬수 있다. Z차례(창문이 놓이는 순서)를 변경할수는 없다. 창문의 Z 차례를 변경하려는 경우에는 *SetWindowPos( )*함수를 사용한다. 아래에 선언을 보여 주었다.

```
BOOL SetWindowPos(HWND hwnd, HWND hWhere,
                  int NewX, int NewY, int NewWidth,
                  int NewHeight, UINT How);
```

hwnd에 처리의 대상으로 되는 창문의 손잡이를 설정한다. hWhere에는 창문의 Z차례를 설정한다. hWhere에는 hwnd에서 지정된 창문의 밑에 놓이는 창문의 손잡이 혹은 Z차례의 절대위치를 가리키는 값을 설정한다.

실례로 hwnd에서 지정된 창문을 제일 우에 배치하려면 HWND_TOP를 설정한다. 창문을 제일 밑에 배치한다면 HWND_BOTTOM을 설정한다. 왼쪽웃모서리의 자리표를 NewX와 NewY에 설정한다. 새로운 너비와 높이를 NewWidth와 NewHeight에 설정한다.

How의 값은 창문의 위치를 변경할 때의 상태를 지정한다. 실례로 SWP_HIDEWINDOW라면 창문이 비표시로 되며 WSP_SHOWWINDOW라면 창문이 표시된다. SetWindowPos( )는 호출이 성공하면 령 아닌 값을 돌려 주고 실패하면 령을 돌려 준다. NewX와 NewY에 설정한 자리표는 창문이 제일 웃준위의 창문인가 새끼창문인가에 따라 다른 의미를 가진다. 새끼창문의 경우는 어미창문의 의뢰자구역에 대한 자리표로 된다. 제일 웃준위의 창문인 경우는 화면에 대한 자리표로 된다.

MoveWindow( )와 SetWindowsPos( )는 편리한 함수로서 그것들을 활용하여 프로그람작성에서 제기되는 여러가지 문제들을 해결할수 있다.

머리부조종체에 머리부항목을 삽입하기

머리부조종체에 적절한 크기를 설정해 주면 거기에 *머리부항목을 삽입*할수 있다. 매 머리부항목은 한개의 렬을 정의하게 된다. 머리부항목을 삽입하려면 먼저 *HDITEM* 구조체에 머리부항목의 정보를 설정하고 다음 HeaderInsertItem( )함수를 호출한다.

례를 들어 아래의 프로그람코드는 머리부조종체의 첫렬로 되는 머리부항목을 삽입하는 코드이다.

```
HDITEM hditem;

hditem.mask = HDI_FORMAT | HDI_WIDTH | HDI_TEXT;
hditem.pszText = "Heading #1";
hditem.cchTextMax = strlen(hditem.pszText);
hditem.cxy = 100;
hditem.fmt = HDF_STRING | HDF_LEFT;
Header_InsertItem(hHeadWnd, 0, &hditem);
```

이 프로그람코드는 렬에 [Heading #1]라는 문자렬을 표시한다. 문자렬은 왼쪽맞추기된다. 머리부항목의 너비에는 100이라는 적당한 값을 설정하고 있다.

머리부항목을 삽입할 때는 머리부조종체에 의해 관리되는 색인(여기에서는 0 으로 되여 있다.)을 설정하여야 한다는 점을 주의해야 한다.

머리부조종체의 표시

필요한 머리부항목을 모두 삽입하였다면 ShowWindow( )함수를 호출하여 매개 머리부조종체를 표시할수 있다. 례를 들어 hHeadWnd 라는 창문손잡이의 머리부조종체를 표시한다면 아래와 같다.

```
ShowWindow(hHeadWnd, SW_SHOW); // 머리부조종체를 표시한다.
```

머리부조종체에 표시되는것은 매 렬의 제목으로 되는 머리부항목뿐이라는데 주의해야 한다. 이 시점에서는 머리부조종체밑의 자료가 표시되지 않는다. 자료들을 표시하기 위한 처리는 프로그람의 다른 부분에서 진행하게 된다. 그보다 앞서 머리부조종체가 생성하는 여러가지 통지문들을 파악하여야 한다.

머리부조종체의 통지문

머리부조종체는 피동적인것이 아니라 능동적인 조종체이다. 이 조종체는 사용자에 의해 조작되면 통보문을 생성한다. 례를 들어 사용자가 머리부항목의 치수를 변경하면 그것을 알려 주는 통보문이 프로그람에 전송된다. 통보문에 응답하여 적절한 처리를 진

행해야 한다. 머리부조종체는 몇 종류의 통보문을 생성하며 어느 통보문을 처리해야 하는가는 프로그람의 목적에 따라 결정되게 된다.

머리부조종체는 WM_NOTIFY 통보문을 리용하여 프로그람에 통보문을 보낸다. WM_NOTIFY 통보문이 보내졌을 때 wParam에는 통보문을 생성한 조종체의 ID가 보관되여 있다. 머리부조종체의 대부분의 통보문에서는 lParam 에 *NMHEADER 구조체*의 지시자가 보관되여 있다. NMHEADER 구조체의 정의를 아래에 보여 주었다.

```
typedef struct tagNMHEADER
{
     NMHDR hdr;
     int iItem;
     int iButton;
     HDITEM *pitem;
}    NMHEADER;
```

hdr 는 지금까지 몇번 사용한적이 있는 NMHDR 구조체이다. 머리부조종체의 WM_NOTIFY 통보문에서는 통보문을 생성한 머리부조종체의 손잡이가 hdr.wndFrom 에 보관된다. hdr.idFrom 에는 머리부조종체의 ID 가 보관된다. hdr.code 에는 사건의 종류를 가리키는 통지코드가 보관된다. 머리부조종체에서 흔히 사용되는 통지코드들을 표 14-2 에 보여 주었다.

표 14-2. 머리부조종체의 주요한 통지문들

코 드	의 미
HDN_BEGINTRACK	사용자가 렬머리부의 변경을 개시하였다. 크기변경을 허가하는 경우에는 령을 돌려 주고 허가하지 않은 경우에는 령 아닌 값을 돌려준다.
HDN_DIVIDERDBLCLICK	사용자가 경계선을 마우스로 두번 찰칵했다.
HDN_ENDTRACK	사용자가 렬머리부의 크기변경을 끝냈다.
HDN_ITEMCHANGED	머리부항목이 변경되였다.
HDN_ITMCHANGING	머리부항목이 변경되려고 하고 있다. 변경을 허가하는 경우에는 령을 돌려 주고 허가하지 않은 경우에는 령 아닌 값을 돌려 준다.
HDN_ITEMCLICK	사용자가 머리부항목을 찰칵했다.(단추형태의 머리부인 경우에만)
HDN_ITEMDBLCLICK	사용자가 머리부항목을 두번 찰칵했다.(단추형태의 머리부인 경우에만)

HDN_TRACK	사용자가 렬머리부의 크기를 변경중이다. 크기변경을 허가하는 경우에는 령을 돌려 주고 허가하지 않는 경우에는 령 아닌 값을 돌려 준다.

NMHEADER 구조체의 성원 item 은 조작된 머리부항목의 색인을 보여 주었다. iButton 은 눌려진 마우스단추의 종류를 가리킨다. 그러나 대부분의 응용프로그람에서는 이 값을 참조하지 않는다.

pitem 은 조작된 머리부항목을 가리키는 HDITEM 구조체의 지시자이다. 그러나 통지문으로서 *HDN_ITEMCLICK* 혹은 HDN_ITEMDBLCLICK 가 보내 졌을 때는 pitem 의 값이 NULL 로 되여 있다.

이식과 관련한 요점 : NMHEADER 구조체는 이전에 HD_NOTIFY 구조체로 불리워왔다. 낡은 이름의 구조체를 사용하는 번역프로그람도 있다.

머리부조종체의 간단한 실례프로그람

실제로 머리부조종체를 리용하는 간단한 실례프로그람을 작성해 보자. 실례 14-1 의 프로그람은 다섯개의 렬을 가진 머리부조종체를 작성하는 프로그람이다. 이 프로그람에서는 이름과 주소를 보관하는 간단한 주소록을 표시하는데 머리부조종체를 리용하고 있다.

실례 14-1. Head1 프로그람

```
// 머리부조종체의 간단한 실례프로그람

#include <windows.h>
#include <commctrl.h>
#include <cstring>
#include <cstdio>
#include "head.h"

#define NUMCOLS 5
#define DEFWIDTH 100
#define MINWIDTH 20
#define SPACING 8
#define NUMENTRIES 7

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
```

```
HWND InitHeader(HWND hParent);
void InitDatabase(void);

char szWinName[] = "MyWin"; // 창문클라스의 이름

HINSTANCE hInst;
HWND hHeadWnd;

int HeaderHeight;

int columns[NUMCOLS] = {DEFWIDTH, DEFWIDTH,
                        DEFWIDTH, DEFWIDTH,
                        DEFWIDTH};

struct MailList {
  char name[40];
  char street[40];
  char city[40];
  char state[3];
  char code[11];
} data[NUMENTRIES];

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  MSG msg;
  WNDCLASSEX wcl;
  HACCEL hAccel;
  HWND hwnd;
  INITCOMMONCONTROLSEX cc;

  // 창문클라스를 정의한다.
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst;     // 실체의 손잡이
  wcl.lpszClassName = szWinName; // 창문클라스의 이름
  wcl.lpfnWndProc = WindowFunc;  // 창문함수
  wcl.style = 0;                 // 체계설정의 형식
```

```
wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

wcl.lpszMenuName = "HeaderMenu"; // 기본차림표

wcl.cbClsExtra = 0; // 보조기억기령역은 필요 없다.
wcl.cbWndExtra = 0;

// 창문의 배경색을 흰색으로 한다.
wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

// 창문클라스를 등록한다.
if(!RegisterClassEx(&wcl)) return 0;

/* 창문클라스가 등록되였으므로
   창문을 작성할수 있다. */
hwnd = CreateWindow(
  szWinName, // 창문클라스의 이름
  "Using a Header Control", // 제목
  WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
  CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 너비는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 높이는 Windows가 결정하게 한다.
  NULL,          // 어미창문은 없다.
  NULL,          // 차림표는 없다.
  hThisInst,     // 실체의 손잡이
  NULL           // 추가파라메터는 없다.
);

hInst = hThisInst; // 현재의 실체손잡이를 보관한다.

// 건반가속기를 적재한다.
hAccel = LoadAccelerators(hThisInst, "HeaderMenu");

// 공통조종체를 초기화한다.
```

```
  cc.dwSize = sizeof(INITCOMMONCONTROLSEX);
  cc.dwICC = ICC_LISTVIEW_CLASSES;
  InitCommonControlsEx(&cc);

  // 창문을 표시한다.
  ShowWindow(hwnd, nWinMode);
  UpdateWindow(hwnd);

  // 통보문순환고리를 작성한다.
  while(GetMessage(&msg, NULL, 0, 0))
  {
    if(!TranslateAccelerator(hwnd, hAccel, &msg)) {
      TranslateMessage(&msg); // 건반통보를 변환한다.
      DispatchMessage(&msg); // Windows 2000에 조종을 넘긴다.
    }
  }

  return msg.wParam;
}

/* 이 함수는 Windows 2000으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받아 들인다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  int response;
  RECT rect;
  HDLAYOUT layout;
  WINDOWPOS winpos;
  NMHEADER *hdnptr;
  HDITEM *hdiptr;
  PAINTSTRUCT ps;
  TEXTMETRIC tm;
  SIZE size;

  char str[80];
  int i, j, ColStart, chrs;
```

```
int entry;
int linespacing;

HDC hdc;

switch(message) {
  case WM_CREATE:
    hHeadWnd = InitHeader(hwnd);
    InitDatabase( );
    break;
  case WM_COMMAND:
    switch(LOWORD(wParam)) {
      case IDM_EXIT:
        response = MessageBox(hwnd, "Quit the Program?",
                              "Exit", MB_YESNO);
        if(response == IDYES) PostQuitMessage(0);
        break;
      case IDM_HELP:
        MessageBox(hwnd, "Try resizing the header.",
                   "Help", MB_OK);
        break;
    }
    break;
  case WM_SIZE:
    /* 어미창문의 크기가 변경되였을 때
       머리부항목의 크기도 변경한다. */
    GetClientRect(hwnd, &rect);
    layout.prc = &rect;
    layout.pwpos = &winpos;
    Header_Layout(hHeadWnd, &layout);

    MoveWindow(hHeadWnd, winpos.x, winpos.y,
               winpos.cx, winpos.cy, 1);
    break;
  case WM_NOTIFY:
    if(LOWORD(wParam) == ID_HEADCONTROL) {
      hdnptr = (NMHEADER *) lParam;
      hdiptr = (HDITEM *) hdnptr->pitem;
```

```
        switch(hdnptr->hdr.code) {
          case HDN_ENDTRACK: // 사용자가 렬의 너비를 변경했다.
            if(hdiptr->cxy < MINWIDTH) {
              hdiptr->cxy = MINWIDTH;
              columns[hdnptr->iItem] = MINWIDTH;
            }
            else
              columns[hdnptr->iItem] = hdiptr->cxy;
            InvalidateRect(hwnd, NULL, 1);
            break;
        }
      }
      break;
    case WM_PAINT:
      hdc = BeginPaint(hwnd, &ps);

      GetTextMetrics(hdc, &tm);
      linespacing = tm.tmHeight + tm.tmInternalLeading;

      for(entry = 0; entry < NUMENTRIES; entry++) {
        ColStart = 0;
        for(i=0; i<NUMCOLS; i++) {
          switch(i) {
            case 0: strcpy(str, data[entry].name);
              break;
            case 1: strcpy(str, data[entry].street);
              break;
            case 2: strcpy(str, data[entry].city);
              break;
            case 3: strcpy(str, data[entry].state);
              break;
            case 4: strcpy(str, data[entry].code);
              break;
          }

          // 표시되지 않은 부분이 있는 경우는 「...」를 추가한다.
          GetTextExtentPoint32(hdc, str, strlen(str), &size);
          j = 2;
```

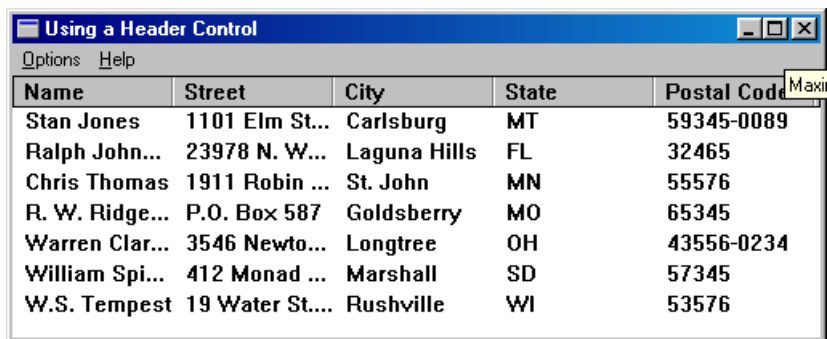
```
            while((columns[i]-SPACING) < size.cx) {
              chrs = columns[i] / tm.tmAveCharWidth;
              strcpy(&str[chrs-j], "...");
              GetTextExtentPoint32(hdc, str, strlen(str), &size);
              j++;
            }

            TextOut(hdc, ColStart+SPACING,
                    HeaderHeight+(entry*linespacing),
                    str, strlen(str));

            ColStart += columns[i];
          }
        }
        EndPaint(hwnd, &ps);
        break;
      case WM_DESTROY: // 프로그람을 끝낸다.
        PostQuitMessage(0);
        break;
      default:
        /* 이 switch 문에서 지정된것 이외의 통보문은
           Windows 2000 에 처리를 맡긴다. */
        return DefWindowProc(hwnd, message, wParam, lParam);
    }

    return 0;
}

// 머리부조종체의 초기화
HWND InitHeader(HWND hParent)
{
    HWND hHeadWnd;
    RECT rect;
    HDLAYOUT layout;
    WINDOWPOS winpos;
    HDITEM hditem;

    GetClientRect(hParent, &rect);
```

```
// 머리부조종체를 작성한다.
hHeadWnd = CreateWindow(WC_HEADER, NULL,
                    WS_CHILD | WS_BORDER,
                    CW_USEDEFAULT, CW_USEDEFAULT,
                    0, 0, hParent,
                    (HMENU) ID_HEADCONTROL,
                    hInst, NULL);

// 의뢰자구역의 크기에 맞는 머리부조종체의 크기를 얻는다.
layout.pwpos = &winpos;
layout.prc = &rect;
Header_Layout(hHeadWnd, &layout);

// 의뢰자구역의 크기에 맞추어 머리부조종체의 크기를 조정한다.
MoveWindow(hHeadWnd, winpos.x, winpos.y,
            winpos.cx, winpos.cy, 0);

HeaderHeight = winpos.cy; // 머리부항목의 높이를 보관한다.

// 머리부항목에 자료를 삽입한다.
hditem.mask = HDI_FORMAT | HDI_WIDTH | HDI_TEXT;
hditem.pszText = "Name";
hditem.cchTextMax = strlen(hditem.pszText);
hditem.cxy = DEFWIDTH;
hditem.fmt = HDF_STRING | HDF_LEFT;
Header_InsertItem(hHeadWnd, 0, &hditem);

hditem.pszText = "Street";
hditem.cchTextMax = strlen(hditem.pszText);
Header_InsertItem(hHeadWnd, 1, &hditem);

hditem.pszText = "City";
hditem.cchTextMax = strlen(hditem.pszText);
Header_InsertItem(hHeadWnd, 2, &hditem);

hditem.pszText = "State";
hditem.cchTextMax = strlen(hditem.pszText);
```

```
    Header_InsertItem(hHeadWnd, 3, &hditem);

    hditem.pszText = "Postal Code";
    hditem.cchTextMax = strlen(hditem.pszText);
    Header_InsertItem(hHeadWnd, 4, &hditem);

    ShowWindow(hHeadWnd, SW_SHOW); // 머리부조종체를 표시한다.

    return hHeadWnd;
}

// 머리부조종체의 밑에 표시되는 실례자료
void InitDatabase(void)
{
    strcpy(data[0].name, "Stan Jones");
    strcpy(data[0].street, "1101 Elm St. S.W.");
    strcpy(data[0].city, "Carlsburg");
    strcpy(data[0].state, "MT");
    strcpy(data[0].code, "59345-0089");

    strcpy(data[1].name, "Ralph Johnson");
    strcpy(data[1].street, "23978 N. Wesley Blvd.");
    strcpy(data[1].city, "Laguna Hills");
    strcpy(data[1].state, "FL");
    strcpy(data[1].code, "32465");

    strcpy(data[2].name, "Chris Thomas");
    strcpy(data[2].street, "1911 Robin Way");
    strcpy(data[2].city, "St. John");
    strcpy(data[2].state, "MN");
    strcpy(data[2].code, "55576");

    strcpy(data[3].name, "R. W. Ridgeway");
    strcpy(data[3].street, "P.O. Box 587");
    strcpy(data[3].city, "Goldsberry");
    strcpy(data[3].state, "MO");
    strcpy(data[3].code, "65345");
```

```
    strcpy(data[4].name, "Warren Clarence");
    strcpy(data[4].street, "3546 Newton Lane");
    strcpy(data[4].city, "Longtree");
    strcpy(data[4].state, "OH");
    strcpy(data[4].code, "43556-0234");

    strcpy(data[5].name, "William Spinoza");
    strcpy(data[5].street, "412 Monad Ave.");
    strcpy(data[5].city, "Marshall");
    strcpy(data[5].state, "SD");
    strcpy(data[5].code, "57345");

    strcpy(data[6].name, "W.S. Tempest");
    strcpy(data[6].street, "19 Water St. Apt 2A");
    strcpy(data[6].city, "Rushville");
    strcpy(data[6].state, "WI");
    strcpy(data[6].code, "53576");
  }
```

이 프로그람은 아래의 자원파일을 필요로 한다.

```
// 머리부조종체의 실례
#include <windows.h>
#include "head.h"

HeaderMenu MENU
{
  POPUP "&Options"
  {
    MENUITEM "E&xit\tCtrl+X", IDM_EXIT
  }
  MENUITEM "&Help", IDM_HELP
}

HeaderMenu ACCELERATORS
{
  "^X", IDM_EXIT
  VK_F1, IDM_HELP, VIRTKEY
}
```

머리부파일 HEAD.H 의 내용을 아래에 보여 주었다. 이 머리부파일에는 이후에 작성하는 프로그람에서 사용되는 값들도 들어 있다.

```
#define IDM_EXIT            100
#define IDM_HELP            101
#define IDM_RESET           102

#define ID_HEADCONTROL      500
```

프로그람의 실행결과를 그림 14-1 에 보여 주었다.

Using a Header Control

Options Help

Name	Street	City	State	Postal Code
Stan Jones	1101 Elm St...	Carlsburg	MT	59345-0089
Ralph John...	23978 N. W...	Laguna Hills	FL	32465
Chris Thomas	1911 Robin ...	St. John	MN	55576
R. W. Ridge...	P.O. Box 587	Goldsberry	MO	65345
Warren Clar...	3546 Newto...	Longtree	OH	43556-0234
William Spi...	412 Monad ...	Marshall	SD	57345
W.S. Tempest	19 Water St....	Rushville	WI	53576

그림 14-1. 머리부조종체의 첫 실례프로그람의 실행결과

머리부조종체의 첫 실례프로그람의 상세

프로그람이 WM_CREATE 통보문을 받았을 때 InitHeader( ) 및 InitDatabase( ) 라는 두 함수가 호출되고 있다. InitHeader( )는 머리부조종체의 작성과 초기화를 진행한다.

함수의 파라메터로 어미창문의 손잡이를 넘긴다. 함수는 돌림값으로서 머리부조종체의 손잡이를 돌려 준다. 머리부조종체의 ID 는 ID_HEADCONTROL 로 하고 있다. 이 ID는 프로그람의 자원파일에 정의되여 있으며 WM_NOTIFY 통보문을 받았을 때 머리부조종체를 식별하기 위한것이다.(다른 종류의 조종체들도 WM_NOTIFY 통보문을 보내오기때문이다.)

이 함수에서 진행되고 있는 다른 처리는 지금까지 설명해 온것들이므로 설명은 략한다. 이 실례프로그람에서는 모든 렬에 체계설정으로서 100 이라는 동일한 크기를 주고 있다. 물론 프로그람의 실행시에 사용자가 렬의 크기를 변경할수도 있다. 매 렬의 크기는 배렬 columns 에 보관된다.

머리부조종체의 높이가 HeaderHeight 라는 대역변수에 보관되여 있는 점에 주의해

야 한다. 머리부조종체는 어미창문의 의뢰자구역의 상단에 배치되므로 기본창문에 정보를 표시할 때의 편위로서 HeaderHeight 의 값이 리용된다.

머리부조종체의 작성과 초기화가 끝나면 InitDatabase( )를 사용하여 주소록의 초기화가 진행된다. 이 자료기지는 MailList 구조체의 배렬 data 에 보관된다. MailList 구조체는 이름과 주소의 정보를 보관하는 문자렬의 배렬로 구성되여 있다. *WM_PAINT* 통보문을 받았을 때 배렬 data 의 내용이 표시된다.

이 프로그람에서 처리하고 있는 머리부조종체의 통보문은 *HDN_ENDTRACK* 만이다. 이 통보문은 사용자가(우측의 경계선을 끌기하여) 렬머리부의 크기변경을 끝냈을 때 발송된다.

렬의 너비가 변경되면 그에 맞추어 머리부조종체의 밑에 표시된 정보의 표시도 변경하여야 한다. 그러기 위해 columns 를 새로운 값으로 변경하고 다시그리기를 진행한다. 사용자는 렬을 MINWIDTH 에 설정된 너비보다 작게 할수 없다는데도 주의해야 한다. 이것은 항상 머리부가 표시되도록 보증한다.

WM_PAINT 의 case 문에도 주목해야 할 부분이 있다. 그것은 렬에 표시되는 정보가 현재렬의 너비보다 긴 경우는 정보가 잘리우고 마감에 생략기호(...)가 추가된다는것이다. 이렇게 하여 현재 표시되여 있는 정보외에도 남은 부분이 있다는것을 사용자에게 알려 줄수 있다.

이 프로그람에는 또 한가지 흥미 있는 점이 있다. *WM_SIZE* 의 case 문을 보면 알수 있지만 창문의 크기가 변경되면 WM_SIZE 통보문이 발송된다. 이 프로그람에서는 WM_SIZE 통보문에 대한 응답으로서 어미창문의 크기에 맞추어 머리부조종체의 크기를 조정하고 있다.

그러나 이 처리는 반드시 필요한것은 아니다. 실제로는 머리부조종체의 크기를 고정시키는 응용프로그람들도 있다. 이 실례프로그람에서는 머리부조종체의 크기를 어미창문에 맞추어 조정하는 방법을 보여 주고 있다.

머리부조종체의 고급한 사용법

앞의 실례프로그람은 머리부조종체를 작성하는데 필요되는 기본적인 기술만을 보여주는것이였다. 머리부조종체가 가지고 있는 다양한 기능을 활용하면 프로그람을 보다 강력하고 리용하기 쉬운것으로 만들수 있다. 여기에서는 머리부조종체의 고급한 기능을 몇가지 사용해 보기로 한다. 구체적으로는 앞서 본 프로그람에 아래의 기능들을 추가한다.

- 단추형태의 머리부항목을 사용하며 마우스사건에 응답할수 있게 한다.
- 단추형태의 머리부항목이 찰칵되였을 때 렬의 너비를 두배로 한다
- 단추형태의 머리부항목이 두번 찰칵되였을 때 렬의 너비를 본래 크기로 복귀한다.

● 머리부항목의 크기가 변경되였을 때 그와 동시에 머리부항목의 밑에 표시된 정보를 넓게 표시하거나 좁게 표시한다.

이러한 기능들을 실현하기 위한 방법을 설명하자.

단추형태의 머리부항목의 작성

단추형태의 머리부항목을 작성하는것은 아주 간단하다. 머리부조종체를 작성할 때의 형식에 *HDS_BUTTONS* 를 포함시키는것뿐이다. 이 설정을 진행하면 매 머리부항목이 누름단추로 되며 마우스사건에 응답할수 있게 된다. 표준적인 머리부항목의 속성은 그대로 계승된다. 례를 들면 경계선을 끌기하여 단추형태의 머리부항목의 크기를 변경할수도 있다.

마우스사건에 대한 응답

단추형태의 머리부항목이 마우스로 찰칵되면 *HDN_ITEMCLICK* 통보문이 발송된다. 단추가 두번 찰칵되면 *HDN_ITEMDBLCLICK* 통보문이 발송된다. 이런 통지문들을 받은 프로그람은 필요에 따라 여러가지 처리를 진행한다. 특별히 처리해야 할것은 없다.

HDN_TRACK 통보문

사용자가 머리부항목의 크기변경을 개시하면 머리부조종체가 *HDN_BEGINTRACK* 통보문을 생성한다. 사용자가 크기변경을 완료하면 머리부조종체가 *HDN_ENDTRACK* 통보문을 생성한다. 크기의 변경중에 머리부조종체가 *HDN_TRACK* 통보문을 생성한다. 프로그람에서 이 통보문들을 받으면 렬에 표시되여 있는 정보의 크기를 변경하여야 한다. 이렇게 하면 사용자는 크기변경의 효과를 직접 볼수 있다.

머리부조종체의 확장된 실례프로그람

실례 14-2의 프로그람은 앞에서 작성한 머리부조종체의 실례프로그람에 우에서 설명한 확장기능들을 추가한것이다.

실례 14-2. Head2 프로그람

```
// 머리부조종체의 확장된 실례프로그람

#include <windows.h>
#include <commctrl.h>
#include <cstring>
#include <cstdio>
#include "head.h"
```

```
#define NUMCOLS 5
#define DEFWIDTH 100
#define MINWIDTH 20
#define MAXWIDTH 400
#define SPACING 8
#define NUMENTRIES 7

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
HWND InitHeader(HWND hParent);
void InitDatabase(void);

char szWinName[] = "MyWin"; // 창문클라스의 이름

HINSTANCE hInst;
HWND hHeadWnd;

int HeaderHeight;

int columns[NUMCOLS] = {DEFWIDTH, DEFWIDTH,
                        DEFWIDTH, DEFWIDTH,
                        DEFWIDTH};

struct MailList {
  char name[40];
  char street[40];
  char city[40];
  char state[3];
  char code[11];
} data[NUMENTRIES];

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  MSG msg;
  WNDCLASSEX wcl;
  HACCEL hAccel;
  HWND hwnd;
```

```
INITCOMMONCONTROLSEX cc;

// 창문클라스를 정의한다.
wcl.cbSize = sizeof(WNDCLASSEX);

wcl.hInstance = hThisInst;      // 실체의 손잡이
wcl.lpszClassName = szWinName; // 창문클라스의 이름
wcl.lpfnWndProc = WindowFunc;  // 창문함수
wcl.style = 0;                 // 체계설정의 형식

wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

wcl.lpszMenuName = "HeaderMenuEnhanced"; // 기본차림표

wcl.cbClsExtra = 0; // 보조기억기렁역은 필요 없다.
wcl.cbWndExtra = 0;

// 창문의 배경색을 흰색으로 한다.
wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

// 창문클라스를 등록한다.
if(!RegisterClassEx(&wcl)) return 0;

/* 창문클라스가 등록되였으므로
   창문을 작성할수 있다. */
hwnd = CreateWindow(
  szWinName, // 창문클라스의 이름
  "An Enhanced Header Control", // 제목
  WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
  CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 너비는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 높이는 Windows가 결정하게 한다.
  NULL,          // 어미창문은 없다.
  NULL,          // 차림표는 없다.
  hThisInst,     // 실체의 손잡이
```

```
        NULL            // 추가파라메터는 없다.
      );

      hInst = hThisInst; // 현재의 실체손잡이를 보관한다.

      // 건반가속기를 적재한다.
      hAccel = LoadAccelerators(hThisInst, "HeaderMenuEnhanced");

      // 공통조종체를 초기화한다.
      cc.dwSize = sizeof(INITCOMMONCONTROLSEX);
      cc.dwICC = ICC_LISTVIEW_CLASSES;
      InitCommonControlsEx(&cc);

      // 창문을 표시한다.
      ShowWindow(hwnd, nWinMode);
      UpdateWindow(hwnd);

      // 통보문순환고리를 작성한다.
      while(GetMessage(&msg, NULL, 0, 0))
      {
        if(!TranslateAccelerator(hwnd, hAccel, &msg)) {
          TranslateMessage(&msg); // 건반통보를 변환한다.
          DispatchMessage(&msg); // Windows 2000에 조종을 넘긴다.
        }
      }

      return msg.wParam;
    }

    /* 이 함수는 Windows 2000으로부터 호출되여
       통보문대기렬에서 꺼낸 통보문을 받아 들인다.
    */
    LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                                WPARAM wParam, LPARAM lParam)
    {
      int response;
      RECT rect;
      HDLAYOUT layout;
```

```
WINDOWPOS winpos;
NMHEADER *hdnptr;
HDITEM *hdiptr, hditem;
PAINTSTRUCT ps;
TEXTMETRIC tm;
SIZE size;

char str[80];
int i, j, ColStart, chrs;
int entry;
int linespacing;

HDC hdc;

switch(message) {
  case WM_CREATE:
    hHeadWnd = InitHeader(hwnd);
    InitDatabase( );
    break;
  case WM_COMMAND:
    switch(LOWORD(wParam)) {
      case IDM_RESET: // 체계설정의 너비로 복귀한다.
        hditem.mask = HDI_WIDTH;
        for(i=0; i<NUMCOLS; i++) {
          Header_GetItem(hHeadWnd, i, &hditem);
          hditem.cxy = DEFWIDTH;
          columns[i] = DEFWIDTH;
          Header_SetItem(hHeadWnd, i, &hditem);
        }
        InvalidateRect(hwnd, NULL, 1);
        break;
      case IDM_EXIT:
        response = MessageBox(hwnd, "Quit the Program?",
                              "Exit", MB_YESNO);
        if(response == IDYES) PostQuitMessage(0);
        break;
      case IDM_HELP:
        MessageBox(hwnd, "Try resizing the header.",
```

```
                    "Help", MB_OK);
            break;
        }
        break;
    case WM_SIZE:
        /* 어미창문의 크기가 변경되였을 때
           머리부조종체의 크기도 변경한다. */
        GetClientRect(hwnd, &rect);
        layout.prc = &rect;
        layout.pwpos = &winpos;
        Header_Layout(hHeadWnd, &layout);

        MoveWindow(hHeadWnd, winpos.x, winpos.y,
                   winpos.cx, winpos.cy, 1);
        break;
    case WM_NOTIFY:
        if(LOWORD(wParam) == ID_HEADCONTROL) {
          hdnptr = (NMHEADER *) lParam;
          hdiptr = (HDITEM *) hdnptr->pitem;
          switch(hdnptr->hdr.code) {
            case HDN_ENDTRACK: // 사용자가 렬의 너비를 변경했다.
              if(hdiptr->cxy < MINWIDTH) {
                hdiptr->cxy = MINWIDTH;
                columns[hdnptr->iItem] = MINWIDTH;
              }
              else
                columns[hdnptr->iItem] = hdiptr->cxy;
              InvalidateRect(hwnd, NULL, 1);
              break;
            case HDN_TRACK: // 사용자가 렬의 너비를 변경하는중이다.
              GetClientRect(hwnd, &rect);
              if(hdiptr->cxy < MINWIDTH) {
                hdiptr->cxy = MINWIDTH;
                columns[hdnptr->iItem] = MINWIDTH;
              }
              else
                columns[hdnptr->iItem] = hdiptr->cxy;
              rect.top = HeaderHeight;
```

```
        InvalidateRect(hwnd, &rect, 1);
        break;
      case HDN_ITEMDBLCLICK: // 사용자가 머리부단추를 두번 찰칵했다.
        // 체계설정의 너비로 복귀한다.
        hditem.mask = HDI_WIDTH;
        Header_GetItem(hHeadWnd, hdnptr->iItem, &hditem);
        hditem.cxy = DEFWIDTH;
        columns[hdnptr->iItem] = DEFWIDTH;
        Header_SetItem(hHeadWnd, hdnptr->iItem, &hditem);
        InvalidateRect(hwnd, NULL, 1);
        break;
      case HDN_ITEMCLICK: // 사용자가 머리부단추를 찰칵했다.
        // 단추의 너비를 두배로 한다.
        hditem.mask = HDI_WIDTH;
        Header_GetItem(hHeadWnd, hdnptr->iItem, &hditem);
        hditem.cxy += hditem.cxy;
        if(hditem.cxy > MAXWIDTH) hditem.cxy = MAXWIDTH;
        columns[hdnptr->iItem] = hditem.cxy;
        Header_SetItem(hHeadWnd, hdnptr->iItem, &hditem);
        InvalidateRect(hwnd, NULL, 1);
        break;
    }
  }
  break;
case WM_PAINT:
  hdc = BeginPaint(hwnd, &ps); // 장치상황을 얻는다.

  GetTextMetrics(hdc, &tm);
  linespacing = tm.tmHeight + tm.tmInternalLeading;

  for(entry = 0; entry < NUMENTRIES; entry++) {
    ColStart = 0;
    for(i=0; i<NUMCOLS; i++) {
      switch(i) {
        case 0: strcpy(str, data[entry].name);
          break;
        case 1: strcpy(str, data[entry].street);
          break;
```

```
            case 2: strcpy(str, data[entry].city);
              break;
            case 3: strcpy(str, data[entry].state);
              break;
            case 4: strcpy(str, data[entry].code);
              break;
          }

          // 표시되지 않은 부분이 있는 경우는 「...」을 추가한다.
          GetTextExtentPoint32(hdc, str, strlen(str), &size);
          j = 2;
          while((columns[i]-SPACING) < size.cx) {
            chrs = columns[i] / tm.tmAveCharWidth;
            strcpy(&str[chrs-j], "...");
            GetTextExtentPoint32(hdc, str, strlen(str), &size);
            j++;
          }

          TextOut(hdc, ColStart+SPACING,
                  HeaderHeight+(entry*linespacing),
                  str, strlen(str));

          ColStart += columns[i];
        }
      }
      EndPaint(hwnd, &ps); // 장치상황을 해제한다.
      break;
    case WM_DESTROY: // 프로그람을 끝낸다.
      PostQuitMessage(0);
      break;
    default:
      /* 이 switch 문에서 지정된것 이외의 통보문은
         Windows 2000 에 처리를 맡긴다. */
      return DefWindowProc(hwnd, message, wParam, lParam);
  }
  return 0;
}
```

```
// 머리부조종체의 초기화
HWND InitHeader(HWND hParent)
{
  HWND hHeadWnd;
  RECT rect;
  HDLAYOUT layout;
  WINDOWPOS winpos;
  HDITEM hditem;

  GetClientRect(hParent, &rect);

  // 머리부조종체를 작성한다.
  hHeadWnd = CreateWindow(WC_HEADER, NULL,
                 WS_CHILD | WS_BORDER | HDS_BUTTONS,
                 CW_USEDEFAULT, CW_USEDEFAULT,
                 0, 0, hParent,
                 (HMENU) ID_HEADCONTROL,
                 hInst, NULL);

  // 의뢰자구역의 크기에 맞는 머리부조종체의 크기를 얻는다.
  layout.pwpos = &winpos;
  layout.prc = &rect;
  Header_Layout(hHeadWnd, &layout);

  // 의뢰자구역의 크기에 맞추어 머리부조종체의 크기를 조정한다.
  MoveWindow(hHeadWnd, winpos.x, winpos.y,
           winpos.cx, winpos.cy, 0);

  HeaderHeight = winpos.cy; // 머리부항목의 높이를 보관한다.

  // 머리부항목에 자료를 삽입한다.
  hditem.mask = HDI_FORMAT | HDI_WIDTH | HDI_TEXT;
  hditem.pszText = "Name";
  hditem.cchTextMax = strlen(hditem.pszText);
  hditem.cxy = DEFWIDTH;
  hditem.fmt = HDF_STRING | HDF_LEFT;
  Header_InsertItem(hHeadWnd, 0, &hditem);
```

```
    hditem.pszText = "Street";
    hditem.cchTextMax = strlen(hditem.pszText);
    Header_InsertItem(hHeadWnd, 1, &hditem);

    hditem.pszText = "City";
    hditem.cchTextMax = strlen(hditem.pszText);
    Header_InsertItem(hHeadWnd, 2, &hditem);

    hditem.pszText = "State";
    hditem.cchTextMax = strlen(hditem.pszText);
    Header_InsertItem(hHeadWnd, 3, &hditem);

    hditem.pszText = "Postal Code";
    hditem.cchTextMax = strlen(hditem.pszText);
    Header_InsertItem(hHeadWnd, 4, &hditem);

    ShowWindow(hHeadWnd, SW_SHOW); // 머리부조종체를 표시한다.

    return hHeadWnd;
  }

  // 머리부조종체의 밑에 표시되는 실례자료
  void InitDatabase(void)
  {
    strcpy(data[0].name, "Stan Jones");
    strcpy(data[0].street, "1101 Elm St. S.W.");
    strcpy(data[0].city, "Carlsburg");
    strcpy(data[0].state, "MT");
    strcpy(data[0].code, "59345-0089");

    strcpy(data[1].name, "Ralph Johnson");
    strcpy(data[1].street, "23978 N. Wesley Blvd.");
    strcpy(data[1].city, "Laguna Hills");
    strcpy(data[1].state, "FL");
    strcpy(data[1].code, "32465");

    strcpy(data[2].name, "Chris Thomas");
    strcpy(data[2].street, "1911 Robin Way");
```

```
    strcpy(data[2].city, "St. John");
    strcpy(data[2].state, "MN");
    strcpy(data[2].code, "55576");

    strcpy(data[3].name, "R. W. Ridgeway");
    strcpy(data[3].street, "P.O. Box 587");
    strcpy(data[3].city, "Goldsberry");
    strcpy(data[3].state, "MO");
    strcpy(data[3].code, "65345");

    strcpy(data[4].name, "Warren Clarence");
    strcpy(data[4].street, "3546 Newton Lane");
    strcpy(data[4].city, "Longtree");
    strcpy(data[4].state, "OH");
    strcpy(data[4].code, "43556-0234");

    strcpy(data[5].name, "William Spinoza");
    strcpy(data[5].street, "412 Monad Ave.");
    strcpy(data[5].city, "Marshall");
    strcpy(data[5].state, "SD");
    strcpy(data[5].code, "57345");

    strcpy(data[6].name, "W.S. Tempest");
    strcpy(data[6].street, "19 Water St. Apt 2A");
    strcpy(data[6].city, "Rushville");
    strcpy(data[6].state, "WI");
    strcpy(data[6].code, "53576");
}
```

이 프로그람은 앞의 프로그람에서 사용한것과 같은 HEADER.H 라는 머리부파일을 사용한다. 이 프로그람이 사용하는 자원파일을 아래에 보여 주었다.

```
// 머리부조종체의 확장된 실례프로그람
#include <windows.h>
#include "head.h"

HeaderMenuEnhanced MENU
{
```

```
    POPUP "&Options"
    {
      MENUITEM "&Reset\tF2", IDM_RESET
      MENUITEM "E&xit\tCtrl+X", IDM_EXIT
    }
    MENUITEM "&Help", IDM_HELP
  }

  HeaderMenuEnhanced ACCELERATORS
  {
    VK_F2, IDM_RESET, VIRTKEY
    "^X", IDM_EXIT
    VK_F1, IDM_HELP, VIRTKEY
  }
```

이 프로그람의 실행결과를 그림 14-2에 보여 주었다.

An Enhanced Header Control

Options Help

Name	Street	City	State	Postal Code
Stan Jones	1101 Elm St...	Carlsburg	MT	59345-0089
Ralph John...	23978 N. W...	Laguna Hills	FL	32465
Chris Thomas	1911 Robin ...	St. John	MN	55576
R. W. Ridge...	P.O. Box 587	Goldsberry	MO	65345
Warren Clar...	3546 Newto...	Longtree	OH	43556-0234
William Spi...	412 Monad ...	Marshall	SD	57345
W.S. Tempest	19 Water St....	Rushville	WI	53576

그림 14-2. 머리부조종체의 확장판 실례프로그람의 실행결과

머리부조종체의 확장된 실례프로그람의 상세

이 실례프로그람의 내용은 머리부조종체의 첫 실례프로그람과 대부분 같으므로 리해하기 쉽다. 그러나 일부 변경개소에 주의를 돌려야 한다.

우선 *HDS_BUTTONS* 형식을 지정하여 머리부조종체가 작성되고 있다. 이에 의해 단추형태의 머리부항목이 작성된다. 다음으로 WM_NOTIFY 통보문을 처리하고 있는 부분을 잘 살펴 보아야 한다. 대부분의 변경개소는 이 부분에 있다. 아래에 다시한번 프로그람코드를 보여 주었다.

```
  case WM_NOTIFY:
```

```
    if(LOWORD(wParam) == ID_HEADCONTROL) {
      hdnptr = (NMHEADER *) lParam;
      hdiptr = (HDITEM *) hdnptr->pitem;
      switch(hdnptr->hdr.code) {
        case HDN_ENDTRACK: // 사용자가 렬의 너비를 변경했다.
          if(hdiptr->cxy < MINWIDTH) {
            hdiptr->cxy = MINWIDTH;
            columns[hdnptr->iItem] = MINWIDTH;
          }
          else
            columns[hdnptr->iItem] = hdiptr->cxy;
          InvalidateRect(hwnd, NULL, 1);
          break;
        case HDN_TRACK: // 사용자가 렬의 너비를 변경하는중이다.
          GetClientRect(hwnd, &rect);
          if(hdiptr->cxy < MINWIDTH) {
            hdiptr->cxy = MINWIDTH;
            columns[hdnptr->iItem] = MINWIDTH;
          }
          else
            columns[hdnptr->iItem] = hdiptr->cxy;
          rect.top = HeaderHeight;
          InvalidateRect(hwnd, &rect, 1);
          break;
        case HDN_ITEMDBLCLICK: // 사용자가 머리부단추를 두번 찰칵했다.
          // 체계설정의 너비로 복귀한다.
          hditem.mask = HDI_WIDTH;
          Header_GetItem(hHeadWnd, hdnptr->iItem, &hditem);
          hditem.cxy = DEFWIDTH;
          columns[hdnptr->iItem] = DEFWIDTH;
          Header_SetItem(hHeadWnd, hdnptr->iItem, &hditem);
          InvalidateRect(hwnd, NULL, 1);
          break;
        case HDN_ITEMCLICK: // 사용자가 머리부단추를 찰칵했다.
          // 단추의 너비를 두배로 한다.
          hditem.mask = HDI_WIDTH;
          Header_GetItem(hHeadWnd, hdnptr->iItem, &hditem);
          hditem.cxy += hditem.cxy;
```

```
                if(hditem.cxy > MAXWIDTH) hditem.cxy = MAXWIDTH;
                columns[hdnptr->iItem] = hditem.cxy;
                Header_SetItem(hHeadWnd, hdnptr->iItem, &hditem);
                InvalidateRect(hwnd, NULL, 1);
                break;
            }
        }
        break;
```

매 통지문이 어떻게 처리되는가를 알아 보자.

HDN_ENDTRACK 통보문은 첫 실례프로그람과 같은 방법으로 처리되고 있다.

HDN_TRACK 통보문이 보내 졌을 때 머리부항목의 새로운 너비에 맞추어 렬에 표시된 정보의 너비를 조정하고 있다. 이것은 columns 배렬의 내용을 새로운 렬너비의 값으로 갱신하고 다시그리기를 하여 실현된다.(InvalidateRect( )를 호출한다.)

그러므로 사용자가 경계선을 끌기하면 머리부항목의 밑에 표시된 정보가 동적으로 갱신되므로 머리부항목을 넓히거나 좁힐 때의 효과를 직접 검사할수 있다. 이렇게 하면 사용자는 매 렬에 직접 적절한 크기를 설정할수 있게 되므로 프로그람의 조작기능이 높아 진다.

사용자가 단추형태의 머리부항목을 찰칵하면 HDN_ITEMCLICK 통보문이 발송된다. 프로그람은 이 통보문에 대한 응답으로서 렬의 너비를 두배로 넓힌다. 머리부항목의 정보를 얻기 위하여서는 *Header_GetItem( )*을 사용하고 정보를 설정하려면 *Header_SetItem( )*을 사용해야 한다는 점에 주의를 돌려야 한다.

이러한 처리가 필요하게 되는것은 HDN_ITEMCLICK 통보문이 보내 졌을 때 NMHEADER 구조체의 pitem 성원의 값이 NULL 로 되여 있기때문이다. 다시말하여 pitem 을 참조하는것으로는 찰칵된 머리부항목을 식별할수 없기때문이다.

사용자가 단추형태의 머리부항목을 찰칵하면 *HDN_ITEMDBLCLICK* 통보문이 발송된다. 프로그람은 이 통보문에 대한 응답으로서 머리부항목의 너비를 본래의 크기로 복귀한다.

반복해서 언급하지만 머리부항목의 정보를 얻으려면 Header_GetItem( )을 사용하고 정보를 설정하려면 Header_SetItem( )을 사용해야 한다는 점을 명심해야 한다.

이 프로그람에는 또 하나의 확장기능이 있다. 그것은 사용자가 [Options]차림표에서 [Reset]를 선택하면 모든 렬이 본래의 크기로 복귀되는것이다.

자체로 해보기

렬머리부를 동적으로 추가하거나 삭제할수도 있다. 이것은 정보를 여러가지 형식으로 표시할 때 편리하다. 례를 들면 사용자가 선택한 추가선택항목의 종류에 따라 머리부항목을 적절히 추가하는것과 같은것이 가능하다. 이 기능을 실례프로그람에 추가해 보시오.

또 하나의 도전으로서 사용자가 머리부조종체를 비표시로 할수 있는 가능성이다. 례를 들어 사용자가 렬의 너비를 조정하도록 하기 위해 머리부조종체를 리용하고 그것이 끝나면 머리부조종체를 비표시로 한다. 이렇게 하면 화면에 보다 많은 정보를 표시할수 있게 된다.

월사업표조종체

매우 편리한 조종체의 하나로서 *월사업표조종체*가 있다. 월사업표조종체는 1 개월이상의 달력을 표시하고 날자를 선택할수 있게 한다. 체계설정으로는 현재의 날자가 선택된 상태로 월사업표조종체가 표시된다.

사용자는 달력의 월이나 년을 변경할수 있다. 또한 여러가지 튀여나오기차림표, 오르내리기조종체 및 화살단추가 장비되여 있으므로 사용자는 월사업표조종체를 손 쉽게 조작할수 있다. 월사업표조종체는 매우 우수하게 설계되여 있으므로 여러가지 형식으로 표시할수도 있지만 보통은 체계설정의 형식으로도 충분하다.

월사업표조종체는 날자입력을 필요로 하는 응용프로그람에 우수한 시각적효과를 제공할수 있다.

월사업표조종체의 작성

월사업표조종체를 작성하려면 창문클라스에 *MONTHCAL_CLASS* 를 지정하고 CreateWindow( ) 또는 CreateWindowEx( )를 호출한다. WS_CHILD 형식도 지정한다. *WS_VISIBLE*형식과 *WS_BORDER*형식도 포함시키는것이 일반적이다. 월사업표조종체는 초기의 크기를 령으로서 작성한다. 왜냐하면 프로그람을 실행할 때까지는 조종체의 크기를 결정할수 없기때문이다.(례하면 서체의 종류에 맞추어 크기를 결정한다.)

월사업표조종체를 작성할 때는 형식에 여러가지 추가선택항목을 설정할수 있다. 이 장의 실례프로그람에서는 사용되지 않지만 참고할수 있게 표 14-3 에 월사업표조종체의 형식의 추가선택항목을 보여 주고 있다.

표 14-3. 월사업표조종체의 형식의 추가선택항목

형 식	효 과
MCS_DAYSTATE	달력이 표식이 붙는 특정한 날(휴식일 등)의 정보를 요구하게 된다.
MCS_MULTISELECT	날자의 범위를 선택할수 있게 된다.
MCS_NOTODAY	현재의 날자가 지적되지 않게 된다.
MCS_NOTODAYCIRCLE	현재의 날자가 동그라미로 표시되지 않게 된다.
MCS_WEEKNUMBERS	주의 번호가 표시되게 된다.

월사업표조종체에 통보문을 보내기

월사업표조종체는 여러가지 통보문에 응답한다. 흔히 사용되는 통보문들을 표 14-4 에 보여 주었다. 월사업표조종체에 통보문을 보내려면 통보문을 보내는 측으로 조종체의 손잡이를 설정하고 SendMessage( )를 호출한다. 그러나 통보문을 보내는 마크로를 사용하는것이 보다 간단하다. 표 14-4 에 제시한 통보문들에 대응하는 마크로들을 아래에 보여 주었다.

```
BOOL MonthCal_GetCurSel(HWND hCal, SYSEMTIME *st);
BOOL MonthCal_GetMaxTodayWidth(HWND hCal);
BOOL MonthCal_GetMinRequest(HWND hCal, RECT *rect);
BOOL MonthCal_GetToday(HWND hCal, SYSEMTIME *st);
BOOL MonthCal_SetCurSel(HWND hCal, SYSEMTIME *st);
VOID MonthCal_SetToday(HWND hCal, SYSEMTIME *st);
```

표 14-4. 월사업표조종체의 주요한 통보문

통 보 문	의 미
MCM_GETCURSEL	현재 선택되여 있는 날자를 얻는다. 호출이 성공하면 령 아닌 값이 돌려 지고 실패하면 령이 돌려 진다. wParam 에는 령을 설정한다. lParam 에는 현재 선택되여 있는 날자를 보관하기 위한 SYSTEMTIME 구조체의 지시자를 설정한다.
MCM_GETMAXTODAYWIDTH	[Today]로 표시된 부분의 문자렬의 너비를 돌려 준다. wParam 과 lParam 에는 령을 설정한다.
MCM_GETMINREQRECT	달력을 표시할수 있는 창문의 최소크기를 얻는다. 호출이 성공하면 령 아닌 값이 돌려 지고 실패하면 령이 돌려 진다. wParam 에는 령을 설정한다. lParam 에는 크기를 보관하기 위한 RECT 구조체의 지시자를 설정한다.
MCM_GETTODAY	현재의 날자를 얻는다. 호출이 성공하면 령 아닌 값이 돌려 지고 실패하면 령이 돌려 진다. wParam 에는 령을 설정한다. lParam 에는 현재의 날자를 보관하기 위한 SYSTEMTIME 구조체의 지시자를 설정한다.
MCM_SETCURSEL	날자를 선택한다. 호출이 성공하면 령 아닌 값이 돌려 지고 실패하면 령이 돌려 진다.

	wParam 에는 령을 설정한다. lParam 에는 선택하는 날자를 보관하기 위한 SYSTEMTIME 구조체의 지시자를 설정한다.
MCM_SETTODAY	현재의 날자를 설정한다. 돌림값은 돌려지지 않는다. wParam 에는 령을 설정한다. lParam 에는 현재의 날자를 보관하기 위한 SYSTEMTIME 구조체의 지시자를 설정한다.

날자를 설정하거나 얻으려고 할 때는 날자를 SYSTEMTIME 구조체에 설정한다. *SYSTEMTIME* 구조체의 정의를 아래에 보여 주었다.

```
typedef struct _SYSTEMTIME {
      WORD wYear;             // 년
      WORD wMonth;            // 월(1～12)
      WORD wDayOfWeek;        // 요일(0～6)
      WORD wDay;              // 일(1～31)
      WORD wHour;             // 시
      WORD wMinute;           // 분
      WORD wSecond;           // 초
      WORD wMilliseconds;     // 미리초
}     SYSTEMTIME;
```

SYSTEMTIME 구조체에는 날자와 시간의 두개의 성원이 들어 있다. 월사업표조종체에는 시간에 관한 정보가 제공되여 있지 않지만 내부적으로는 관리되고 있다. 실례로 조종체를 작성하면 현재의 시간정보가 현재의 날자정보에 내부적으로 추가된다.

월사업표조종체의 통지문

월사업표조종체는 기본적으로는 피동적인 조종체이나 WM_NOTIFY 통보문과 함께 아래에 보여 준 통지문들을 생성할수도 있다.

통지문	의　　미
MCN_GETDAYSTATE	날자의 표시방법을 알아 보기 위해 발송된다. 실례로 휴식일을 굵은체로 표시하는것 등
MCN_SELCHANGE	날자의 선택이 변경되였다.
MCN_SELECT	사용자가 날자를 선택하였다.

월사업표의 실례프로그람에서는 우와 같은 통보문들을 사용하지 않는다.

월사업표조종체의 크기를 설정하기

이미 설명한것처럼 월사업표조종체를 작성하는 시점에서는 그의 크기를 령으로 하고 후에 크기를 설정하는것이 일반적이다.

달력을 표시할수 있는 최소의 크기를 얻으려면 월사업표조종체에 *MCM_GETMINREQRECT* 통보문을 보낸다. 돌림값으로서 RECT 구조체에 달력을 표시할수 있는 최소의 크기가 돌려 진다.

계속하여 [Today]로 표시된 부분의 문자렬의 너비를 얻어야 한다. 이 문자렬은 현재의 날자를 표시한다. 월사업표조종체의 너비는 [Today]로 표시된 부분의 문자렬의 너비와 MCM_GETMINREQRECT 통보문에서 얻은 너비가운데서 큰것으로 하여야 한다.(보통 [Today]로 표시된 부분의 문자렬의 너비가 작게 되지만 프로그람의 이식성을 고려한다면 두 너비값을 비교하여야 한다.)

적절한 크기가 결정되면 앞에서 설명한 MoveWindow()등의 함수를 사용하여 월사업표조종체의 크기를 설정한다. 뒤에서 작성하는 실례프로그람에서 이 처리를 진행하는 부분을 아래에 보여 주었다.

```
hCal = CreateWindow(MONTHCAL_CLASS,
                    "Month Calendar", // 사용되지 않는다.
                    WS_BORDER | WS_VISIBLE | WS_CHILD,
                    0, 0, 0, 0,
                    hdwnd, NULL, hInst, NULL);

  // 달력의 크기를 확인한다.
  MonthCal_GetMinReqRect(hCal, &rect); // 최소의 크기를 얻는다.

          // [Today]라고 표시된 부분의 문자렬의 너비를 확인한다.
  todaywidth = MonthCal_GetMaxTodayWidth(hCal);
  if(todaywidth > rect.right) rect.right = todaywidth;

          // 마지막으로 조종체의 크기를 설정한다.
MoveWindow(hCal, 0, 0, rect.right, rect.bottom, 1);
```

이러한 순서로 처리하면 조종체의 크기는 달력과 그것에 대응한 차림표나 조종체를 표시하는데 충분하게 된다.

월사업표조종체의 실례프로그람

실례 14-3 의 프로그람은 월사업표조종체의 사용실례를 보여 주었다. 기본차림표를 선택하여 달력을 능동으로 하거나 현재 선택되여 있는 날자를 표시할수 있다.

달력은 대화칸안에 표시된다. 대화칸에는 달력외에도 [Show Date], [OK] 및 [Cancel]의 세개 단추가 있다. [Show Date]단추를 누르면 현재 선택되여 있는 날자가 통보칸에 표시된다. [OK]단추를 누르면 현재 선택되여 있는 날자가 대역변수 st에 보관된다. [Cancel]단추를 누르면 st의 값이 변경되지 않는다. st의 값은 현재 선택되여 있는 날자를 기본차림표에서 표시하는데 리용된다.

실례 14-3. Cal 프로그람

```
// 월사업표조종체의 실례

#include <windows.h>
#include <commctrl.h>
#include <cstdio>
#include "cal.h"

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK DialogFunc(HWND, UINT, WPARAM, LPARAM);

char szWinName[] = "MyWin"; // 창문클라스의 이름

HINSTANCE hInst;
HWND hCal;
SYSTEMTIME st;
int dateset = 0;

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  MSG msg;
  HWND hwnd;
  WNDCLASSEX wcl;
  HACCEL hAccel;
  INITCOMMONCONTROLSEX cc;

  // 창문클라스를 정의한다.
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst;     // 실체의 손잡이
```

```
wcl.lpszClassName = szWinName; // 창문클라스의 이름
wcl.lpfnWndProc = WindowFunc;  // 창문함수
wcl.style = 0;                 // 체계설정의 형식

wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

wcl.lpszMenuName = "MonthCalMenu"; // 기본차림표

wcl.cbClsExtra = 0; // 보조기억기령역은 필요 없다.
wcl.cbWndExtra = 0;

// 창문의 배경색을 흰색으로 한다.
wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

// 창문클라스를 등록한다.
if(!RegisterClassEx(&wcl)) return 0;

/* 창문클라스가 등록되였으므로
   창문을 작성할수 있다. */
hwnd = CreateWindow(
  szWinName, // 창문클라스의 이름
  "Using a Month Calendar", // 제목
  WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
  CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 너비는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 높이는 Windows가 결정하게 한다.
  NULL,          // 어미창문은 없다.
  NULL,          // 차림표는 없다.
  hThisInst,     // 실체의 손잡이
  NULL           // 추가파라메터는 없다.
);

hInst = hThisInst; // 현재의 실체손잡이를 보관한다.

// 건반가속기를 적재한다.
```

```
  hAccel = LoadAccelerators(hThisInst, "MonthCalMenu");

  // 공통조종체를 초기화한다.
  cc.dwSize = sizeof(INITCOMMONCONTROLSEX);
  cc.dwICC = ICC_DATE_CLASSES;
  InitCommonControlsEx(&cc);

  // 창문을 표시한다.
  ShowWindow(hwnd, nWinMode);
  UpdateWindow(hwnd);

  // 통보문순환고리를 작성한다.
  while(GetMessage(&msg, NULL, 0, 0))
  {
    if(!TranslateAccelerator(hwnd, hAccel, &msg)) {
      TranslateMessage(&msg); // 건반통보를 변환한다.
      DispatchMessage(&msg); // Windows 2000 에 조종을 넘긴다.
    }
  }

  return msg.wParam;
}

/* 이 함수는 Windows 2000 으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받아 들인다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  int response;
  char str[255];

  switch(message) {
    case WM_COMMAND:
      switch(LOWORD(wParam)) {
        case IDM_DIALOG:
          DialogBox(hInst, "MonthCalDB", hwnd, (DLGPROC) DialogFunc);
          break;
```

```
        case IDM_GETDATE:
          if(dateset) {
            sprintf(str, "%d/%d/%d",
                    st.wMonth, st.wDay, st.wYear);
            MessageBox(hwnd, str, "Date Selected", MB_OK);
          } else
            MessageBox(hwnd, "Date not set", "Error", MB_OK);
          break;
        case IDM_EXIT:
          response = MessageBox(hwnd, "Quit the Program?",
                                "Exit", MB_YESNO);
          if(response == IDYES) PostQuitMessage(0);
          break;
        case IDM_HELP:
          MessageBox(hwnd, "Try the calendar.", "Help", MB_OK);
          break;
      }
      break;
    case WM_DESTROY: // 프로그람을 끝낸다.
      PostQuitMessage(0);
      break;
    default:
      /* 이 switch문에서 지정된것 이외의 통보문은
         Windows 2000에 처리를 맡긴다. */
      return DefWindowProc(hwnd, message, wParam, lParam);
  }

  return 0;
}

// 대화함수
BOOL CALLBACK DialogFunc(HWND hdwnd, UINT message,
                         WPARAM wParam, LPARAM lParam)
{
  static HWND hCal;
  RECT rect;
  int todaywidth;
  char str[255];
```

```
  SYSTEMTIME tempst;

  switch(message) {
    case WM_INITDIALOG:
      hCal = CreateWindow(MONTHCAL_CLASS,
                "Month Calendar", // 사용되지 않는다.
                WS_BORDER | WS_VISIBLE | WS_CHILD,
                0, 0, 0, 0,
                hdwnd, NULL, hInst, NULL);

      // 달력의 크기를 설정한다.
      MonthCal_GetMinReqRect(hCal, &rect); // 최소크기를 얻는다.
      todaywidth = MonthCal_GetMaxTodayWidth(hCal);
      if(todaywidth > rect.right) rect.right = todaywidth;
      MoveWindow(hCal, 0, 0, rect.right, rect.bottom, 1);
      return 1;
    case WM_COMMAND:
      switch(LOWORD(wParam)) {
        case ID_GETDATE:
          MonthCal_GetCurSel(hCal, &tempst);
          sprintf(str, "%d/%d/%d", tempst.wMonth,
                  tempst.wDay, tempst.wYear);
          MessageBox(hdwnd, str, "Date Selected", MB_OK);
          return 1;
        case IDOK:
          MonthCal_GetCurSel(hCal, &st);
          dateset = 1;
        case IDCANCEL:
          EndDialog(hdwnd, 0);
          return 1;
      }
  }

  return 0;
}
```

이 프로그람은 아래의 자원파일을 필요로 한다.

```
#include <windows.h>
#include "cal.h"

MonthCalMenu MENU
{
  POPUP "&Calendar" {
    MENUITEM "&Calendar\tF2", IDM_DIALOG
    MENUITEM "&Get Selected Date\tF3", IDM_GETDATE
    MENUITEM "E&xit\tCtrl+X", IDM_EXIT
  }
  MENUITEM "&Help", IDM_HELP
}

MonthCalMenu ACCELERATORS
{
  VK_F2, IDM_DIALOG, VIRTKEY
  VK_F3, IDM_GETDATE, VIRTKEY
  "^X", IDM_EXIT
  VK_F1, IDM_HELP, VIRTKEY
}

MonthCalDB DIALOGEX 18, 18, 150, 78
CAPTION "Demonstrate a Month Calendar"
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
{
  DEFPUSHBUTTON "OK", IDOK, 100, 10, 38, 16
  PUSHBUTTON "Show Date", ID_GETDATE, 100, 30, 38, 16
  PUSHBUTTON "Cancel", IDCANCEL, 100, 50, 38, 16
}
```

머리부파일 CAL.H의 내용을 아래에 보여 주었다.

```
#define IDM_DIALOG          100
#define IDM_GETDATE         101
#define IDM_EXIT            102
#define IDM_HELP            103

#define ID_GETDATE          201
```

프로그람의 실행결과를 그림 14-3 에 보여 주었다.

그림 14-3. 월사업표조종체의 실례프로그람의 실행결과

공통조종체를 마치며

이 장까지의 다섯개 장에서는 가장 많이 사용되는 몇가지 *공통조종체*들에 대해 설명하였다. 이 조종체들의 사용방법을 파악하면 다른 조종체들의 사용방법도 자체로 쉽게 배울수 있다. 공통조종체전반에 공통되는 사용방법을 개괄하면 다음과 같다.

- 작성 및 초기화
- 조종체에 통보문을 보낸다.
- 조종체가 생성한 통보문을 처리한다.

제 10 장을 시작할 때 설명한것처럼 공통조종체들을 사용하면 응용프로그람이 볼 품이 있는 최신의 세련된 양상으로 된다.

제 15 장

스레드에 기초한 다중과제처리

이 장에서는 Windows 2000 이 제공하는 스레드(Thread)에 기초한 다중과제처리기능에 대해 설명한다. 이 책의 서두에서 언급한것처럼 Windows 2000 은 두가지 형식의 *다중과제*처리를 지원하고 있다. 첫번째 형식은 프로세스에 기초한 것으로서 초기 판본의 Windows 에서부터 지원되고 있는 다중처리형식이다. *프로세스*란 간단히 말하면 실행중의 프로그람이다. 프로세스에 기초한 다중과제처리에서는 두개이상의 프로세스를 동시에 실행할수 있다.

다중과제처리의 두번째 형식은 스레드에 기초하고 있다. 스레드란 프로세스내에서 실행되는 프로그람의 흐름이다. Windows 2000 에서는 하나의 프로세스가 적어도 하나의 스레드를 가지며 두개이상의 스레드를 가져도 무방하다. 스레드에 기초한 다중과제처리에서는 한 프로그람에 속하는 두개이상의 부분을 동시에 실행할수 있다. 이 기능을 활용하면 매우 효과적인 프로그람을 작성할수 있다. 왜냐하면 프로그람작성자들이 독립적으로 실행되는 스레드들을 자유롭게 작성할수 있으며 프로그람의 실행을 관리할수 있기때문이다.

스레드에 기초한 다중과제처리방식을 사용함으로써 *동기화*라고 하는 특수한 기능도 필요하게 된다. 동기화는 스레드(또는 프로세스)의 동시실행을 적절히 관리하기 위한 특수한 수법이다. Windows 2000 은 동기화를 지원하기 위한 완전한 부분체계를 제공한다. 이 장에서는 그가운데서 관건적인 기능만을 설명한다.

16bit 의 Windows 3.1 에서는 스레드에 기초한 다중과제처리를 지원하지 않는다.

다중스레드프로그람의 작성

만일 *다중스레드프로그람*을 작성해 본 경험이 없다면 이 장을 읽은 후에 놀라움을 느끼게 될것이다. *스레드*기초의 다중과제처리에서는 프로그람의 실행을 부분에까지 완전히 조종할수 있는 새로운 기능을 여러분의 프로그람에 추가하여 준다. 이 기능에 의해 보다 효과적인 프로그람을 작성할수 있게 된다.

실례로 첫번째 스레드에서 파일을 분류하고 두번째 스레드에서는 기억기자원에서 정보를 얻고 세번째 스레드에서 사용자입력을 처리하는것을 실현할수 있기때문이다. 스레드기초의 다중과제처리에 의해 CPU 시간을 거의 랑비함이 없이 개개의 스레드를 동시에 실행할수 있다.

하나의 프로세스가 적어도 하나의 스레드를 가지고 있다는것은 중요하다. 설명상 필요에 의해 이것을 기본스레드(main thread)라고 부르기로 한다. 기본스레드란 프로그람의 기동시에 작성되는 스레드이다. 기본스레드안에서 다른 스레드를 한개이상 작성할수 있다.

보통은 새로운 스레드가 작성되면 그것이 곧 실행을 개시한다. 그러므로 프로세스는 하나의 스레드를 실행하는것으로 개시되여 한개이상의 스레드를 추가하는것으로 된다.

스레드의 작성

스레드를 작성하자면 *CreateThread( )*라는 API 함수를 사용한다. 선언은 다음과 같다.

```
HANDLE CreateThread(LPSECURITY_ATTRIBUTES lpSecAttr,
                    DWORD dwStackSize,
                    LPTHREAD_START_ROUTINE lpThreadFunc,
                    LPVOID lpParam,
                    DWORD dwFlags,
                    LPDWORD lpdwThreadID);
```

lpSecAttr는 스레드에 주는 보안속성에 대한 지시자이다. lpSecAttr에 NULL을 설정하면 체계설정의 보안서술자가 사용된다.

참고 *: Windows 2000의 보안기능에 대해서는 제 20장에서 설명한다. 이 장에서는 체계설정의 보안서술자만을 사용한다.*

매개 스레드는 각각 전용의 탄창(Stack)을 가진다. dwStadkSize 파라메터에는 새로운 스레드의 탄창크기를 byte 단위로 설정할수 있다. 이 파라메터의 값에 령을 설정하면 스레드를 작성한 스레드와 같은 크기의 탄창이 주어 진다. 이 경우에도 후에 필요에 따라 탄창의 크기를 확대할수 있다.(wStadkSize 에는 령을 설정하는것이 일반적이다.)

스레드의 실행은 스레드를 작성한 프로세스에서 *스레드함수*라고 부르는 함수를 기동하는것으로서 개시된다. 스레드의 실행은 스레드함수가 완료할 때까지 계속된다. 이 함수의 주소 즉 스레드의 입구점(Entry point)을 lpThreadFunc 에 설정한다. 모든 스레드함수들은 다음과 같이 선언되여야 한다.

```
DWORD WINAPI threadfunc(LPVOID param);
```

새로운 스레드에 주려는 파라메터를 CreateThread( )의 lpParam 에 준다. 이 32bit의 값이 스레드함수의 파라메터로 된다. 이 파라메터는 임의의 목적으로 사용된다. 스레드함수는 완료되였을 때의 상태를 돌림값으로 넘겨 준다.

dwFlag 파라메터는 스레드의 실행상태를 결정하는 성원이다. 이 파라메터에 령이 설정된 경우는 곧 스레드의 실행이 개시된다. *CREATE_SUSPEND* 가 설정된 경우는 스레드가 정지상태로 작성되여 실행개시를 대기한다.(뒤에서 설명하는 *ResumeThread( )*를 사용하면 실행을 개시시킬수 있다.)

스레드를 식별하기 위한 ID 가 두배단어형의 지시자로서 lpdwThreadID 에 돌려 진다.

이 함수는 호출이 성공하면 스레드의 손잡이를 돌려 주고 실패하면 NULL 을 돌려준다. 스레드의 손잡이는 *CloseHandle( )*을 사용하여 파괴할수 있지만 그렇게 하지 않아도 어미프로세스가 완료할 때 자동적으로 파괴된다.

스레드의 끝내기

이미 설명한것처럼 스레드의 실행은 입구점으로 되여 있는 스레드함수가 완료하는것으로서 끝난다. 그러나 *TerminateThread( )* 또는 *ExitThread( )*의 어느 함수를 사용하면 프로세스로부터 임의의 시점에서 완료시킬수도 있다. 선언은 다음과 같다.

```
BOOL TerminateThread(HANDLE hThread, DWORD dwStatus);
VOID ExitThread(DWORD dwStatus);
```

TerminateThread( )에서는 완료시키려는 스레드의 손잡이를 hThread 에 설정한다. 아래의 ExitThread( )는 스레드가 자기자신을 도중완료할 때 사용한다. 어느 함수도 dwStatus 에 완료코드를 설정한다. 호출이 성공하면 TerminateThread( )는 령이 아닌 값을 돌려 주며 실패하면 령을 돌려 준다.

*ExitThread( )*를 호출하는것은 스레드함수를 정상완료시키는것과 기능적으로 동등

한것이다. 이것은 탄창이 적절하게 재설정된다는것이다.

TerminateThread( )를 사용하여 스레드를 완료시킨 경우는 스레드가 곧 정지되고 탄창 등의 소거(Cleanup)처리가 진행되지 않는다. TerminateThread( )를 사용하면 중요한 처리를 진행하고 있는 스레드를 정지시켜 버리는 일도 있다. 이러한 리유로부터 스레드함수가 정상완료할 때까지 스레드를 강제완료시키지 않는것이 최량이며 가장 간단한 방법으로 된다. 이 장의 실례프로그람이 대체로 이 방법을 사용한다.

다중스레드의 실례프로그람

실례 15-1 에 제시한 프로그람은 [Demonstrate Thread]차림표를 선택하면 두개의 스레드를 작성하는 프로그람이다. 매개 스레드는 for 순환을 사용하여 5000 회 순환하면서 처리를 진행하며 그의 순환회수를 표시한다. 프로그람을 실행하면 두개의 스레드가 동시에 실행되는 상황을 알수 있다.

실례 15-1. Thread 프로그람

```
// 간단한 다중스레드프로그람

#include <windows.h>
#include <cstring>
#include <cstdio>
#include "thread.h"

#define MAX 5000

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
DWORD WINAPI MyThread1(LPVOID param);
DWORD WINAPI MyThread2(LPVOID param);

char szWinName[] = "MyWin"; // 창문클라스의 이름

char str[255]; // 표시할 문자렬을 보관한다.

DWORD Tid1, Tid2; // 스레드 ID

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HWND hwnd;
```

```
MSG msg;
WNDCLASSEX wcl;
HACCEL hAccel;

// 창문클라스를 정의한다.
wcl.cbSize = sizeof(WNDCLASSEX);

wcl.hInstance = hThisInst;      // 실체의 손잡이
wcl.lpszClassName = szWinName; // 창문클라스의 이름
wcl.lpfnWndProc = WindowFunc;  // 창문함수
wcl.style = 0;                 // 체계설정의 형식

wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

wcl.lpszMenuName = "ThreadMenu"; // 기본차림표

wcl.cbClsExtra = 0; // 보조기억기령역은 필요 없다.
wcl.cbWndExtra = 0;

// 창문의 배경색을 흰색으로 한다.
wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

// 창문클라스를 등록한다.
if(!RegisterClassEx(&wcl)) return 0;

/* 창문클라스가 등록되였으므로
   창문을 작성할수 있다. */
hwnd = CreateWindow(
  szWinName, // 창문클라스의 이름
  "Demonstrate Threads", // 제목
  WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
  CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 너비는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 높이는 Windows가 결정하게 한다.
  NULL,          // 어미창문은 없다.
```

```
    NULL,           // 클라스차림표의 덧쓰기는 하지 않는다.
    hThisInst,      // 실체의 손잡이
    NULL            // 추가파라메터는 없다.
  );

  // 건반가속기를 적재한다.
  hAccel = LoadAccelerators(hThisInst, "ThreadMenu");

  // 창문을 표시한다.
  ShowWindow(hwnd, nWinMode);
  UpdateWindow(hwnd);

  // 통보문순환고리를 작성한다.
  while(GetMessage(&msg, NULL, 0, 0))
  {
    if(!TranslateAccelerator(hwnd, hAccel, &msg)) {
      TranslateMessage(&msg); // 건반통보를 변환한다.
      DispatchMessage(&msg); // Windows 2000 에 조종을 넘긴다.
    }
  }

  return msg.wParam;
}

/* 이 함수는 Windows 2000 으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받아 들인다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  int response;

  switch(message) {
    case WM_COMMAND:
      switch(LOWORD(wParam)) {
        case IDM_THREAD: // 스레드를 작성한다.
          CreateThread(NULL, 0,
                       (LPTHREAD_START_ROUTINE)MyThread1,
```

```
                    (LPVOID) hwnd, 0, &Tid1);
            CreateThread(NULL, 0,
                    (LPTHREAD_START_ROUTINE)MyThread2,
                    (LPVOID) hwnd, 0, &Tid2);
            break;
          case IDM_EXIT:
            response = MessageBox(hwnd, "Quit the Program?",
                            "Exit", MB_YESNO);
            if(response == IDYES) PostQuitMessage(0);
            break;
          case IDM_HELP:
            MessageBox(hwnd,
                      "F1: Help\nF2: Demonstrate Threads",
                      "Help", MB_OK);
            break;
        }
        break;
      case WM_DESTROY: // 프로그램을 끝낸다.
        PostQuitMessage(0);
        break;
      default:
        /* 이 switch 문에서 지정된것 이외의 통보문은
           Windows 2000 에 처리를 맡긴다. */
        return DefWindowProc(hwnd, message, wParam, lParam);
    }
    return 0;
}

// 프로세스안에서 실행되는 스레드
DWORD WINAPI MyThread1(LPVOID param)
{
    int i;
    HDC hdc;

    for(i=0; i<MAX; i++) {
      sprintf(str, "Thread 1: loop # %5d ", i);
      hdc = GetDC((HWND) param);
      TextOut(hdc, 1, 1, str, strlen(str));
```

```
    ReleaseDC((HWND) param, hdc);
  }
  return 0;
}

// 프로세스안에서 실행되는 또 하나의 스레드
DWORD WINAPI MyThread2(LPVOID param)
{
  int i;
  HDC hdc;

  for(i=0; i<MAX; i++) {
    sprintf(str, "Thread 2: loop # %5d ", i);
    hdc = GetDC((HWND) param);
    TextOut(hdc, 1, 20, str, strlen(str));
    ReleaseDC((HWND) param, hdc);
  }
  return 0;
}
```

이 프로그람에서 리용하는 자원파일 THREAD.H의 내용은 다음과 같다.

```
#define IDM_THREAD         100
#define IDM_HELP           101
#define IDM_EXIT           102
```

이 프로그람은 아래의 자원파일도 필요로 한다.

```
#include <windows.h>
#include "thread.h"

ThreadMenu MENU
{
  POPUP "&Threads" {
    MENUITEM "Demonstrate &Threads\tF2", IDM_THREAD
    MENUITEM "E&xit\tCtrl+X", IDM_EXIT
  }
  MENUITEM "&Help", IDM_HELP
```

```
}

ThreadMenu ACCELERATORS
{
  VK_F1, IDM_HELP, VIRTKEY
  VK_F2, IDM_THREAD, VIRTKEY
  "^X", IDM_EXIT
}
```

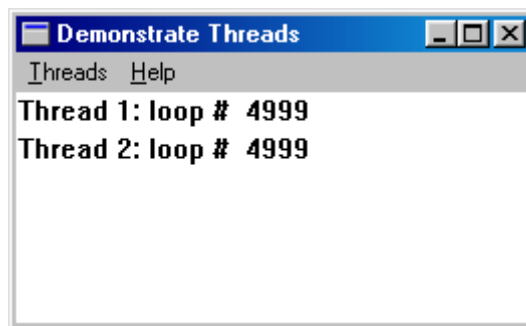



그림 15-1. 다중스레드프로그람의 실행결과

다중스레드프로그람의 상세

[Demonstrate Threads]차림표가 선택되면 다음의 프로그람코드가 실행된다.

```
        case IDM_THREAD: // 스레드를 작성한다.
          CreateThread(NULL, 0,
                      (LPTHREAD_START_ROUTINE)MyThread1,
                      (LPVOID) hwnd, 0, &Tid1);
          CreateThread(NULL, 0,
                      (LPTHREAD_START_ROUTINE)MyThread2,
                      (LPVOID) hwnd, 0, &Tid2);
          break;
```

CreateThread( )의 첫 호출은 MyThread1( )을 기동하며 두번째 호출은 MyThread2( )를 기동한다. 매 스레드함수에는 파라메터로서 기본창문의 창문손잡이(hwnd)가 전달되고 있는 점에 주의해야 한다. 매개 스레드는 이 파라메터로부터 장치상황의 손잡이를 얻고 기본창문에 정보를 표시한다.

실행이 개시되면 매개 스레드(기본스레드를 포함)가 독립적으로 동작한다. 례하면 스레드의 실행도중에 도움말통보칸을 표시할수도 있고 프로그람을 완료할수도 있으며 다

른 스레드를 기동할수도 있다. 프로그람을 완료하면 모든 새끼스레드도 자동적으로 완료한다.

앞으로 더 나가기에 앞서 이 프로그람을 사용하여 여러가지 실험을 해보는것이 유익하다. 실례로 현재는 스레드함수가 완료하는것으로서 스레드가 완료하지만 *ExitThread( )*를 사용하여 스레드를 도중완료시키는것도 시험해 볼수 있다. 매개 스레드의 여러개의 실체(Instance)를 기동하는것도 시험해 볼수 있다.

CreateThread( )와 ExitThread( )의 대용함수

C/C++번역프로그람이나 C/C++의 표준서고의 종류에 따라 *CreateThread( )*와 *ExitThread( )*를 사용하지 않는 편이 좋은 경우가 있다. 그것은 이 두 함수가 약간한 *기억기잃기*를 발생하는 경우가 있기때문이다.

기억기잃기(Memory leak)란 기억기를 잃어 버리는것을 말한다. 이것은 프로그람에 의해 확보된 기억기의 일부가 해제되지 않는것이 원인으로 되여 발생한다.

Microsoft Visual C++를 포함한 많은 번역프로그람들에서는 어느 C/C++표준서고를 리용한 다중스레드프로그람에서 CreateThread( )나 ExitThread( )를 사용한 경우에 기억기잃기가 발생하는 때가 있다.(만일 프로그람에서 C/C++의 표준서고를 리용하지 않는다면 이러한 문제는 발생하지 않는다.) 이 문제를 해결하기 위해 스레드의 개시와 완료에는 Win32 API가 아니라 C/C++의 표준서고함수를 사용하도록 한다.

여기서는 Microsoft가 독자적으로 제공하는 스레드를 작성하는 함수와 완료하는 함수를 소개한다. 만일 다른 번역프로그람을 사용하고 있다면 필요에 따라 사용자지도서를 참고하여 *CreateThread( )*와 ExitThread( )를 사용하여도 문제점이 없는가를 확인하여야 한다.

Microsoft의 독자적인 스레드작성함수와 완료함수

Microsoft Visual C++에 있어서 *CreatThread( )*와 ExitThread( )를 대신하는 함수는 *_beginthreadex( )*와 *_endthreadex( )*이다. 이 함수들은 머리부파일 PROCESS.H에 정의되여 있다. _beginthreadex( )의 선언은 다음과 같다.

```
unsigned long _beginthreadex(void *secAttr, unsigned stackSize,
                    unsigned (__stdcall *threadFunc)(void *),
                    void *param, unsigned flags,
                    unsigned *threadID);
```

보는바와 같이 _beginthreadex( )의 파라메터들은 CreateThread( )와 같다. 또한

매 파라메터의 의미도 동일하다. secAttr는 스레드에 주는 보안속성의 지시자이다. 그러나 secAttr에 NULL을 설정하면 체계설정의 보안서술자가 사용된다.

새로운 스레드의 탄창크기를 byte 단위로 stackSize 파라메터에 설정한다. 이 값에 령을 설정하면 스레드를 작성한 프로세스의 기본스레드와 같은 크기의 탄창이 주어 진다. 이 크기는 후에 필요에 따라 확장할수 있다.

스레드함수(스레드의 입구점)의 주소를 threadFunc 에 설정한다. _beginthreadex ( )에서 스레드함수는 다음과 같이 선언되여야 한다.

```
unsigned __stdcall threadfunc(void *param);
```

이 선언은 CreateThread( )에서 사용되는 스레드함수와 기능적으로는 동일하지만 다른 이름의 자료형이 사용된다. 새로운 스레드에 주려는 파라메터는 CreateThread( )의 lParam 파라메터에 주었던것을 그대로 설정할수 있다.

flags 파라메터는 스레드의 실행상태를 결정하기 위한것이다. 이 파라메터에 령을 설정하면 스레드의 실행이 곧 개시된다. *CREATE_SUSPEND*를 설정하면 스레드가 정지상태에서 작성되여 실행을 대기한다.(ResumeThread( )를 사용하면 실행을 개시할수 있다.) 스레드를 식별하는 ID가 2배단어의 지시자로 threadID에 돌려 진다.

이 함수는 호출이 성공하면 스레드의 손잡이를 돌려 주며 실패하면 령을 돌려 준다.

_endthreadex( )의 선언은 다음과 같다.

```
void _endthreadex(unsigned status);
```

이 함수는 ExitThread( )와 완전히 동일하게 스레드를 정지하는 기능을 가지며 status에는 완료코드를 설정한다.

_beginthreadex( )와 _exitthreadex( )을 사용할 때는 반드시 다중스레드대응서고를 련결하는 설정을 해야 한다.

참고 *: Microsoft 는 스레드의 작성과 완료를 진행하는 _beginthread( ) 및 _endthread( )라는 함수도 제공하고 있다. _beginthread( )는 기본적인 기능만을 가진 함수이며 CreateThread( )에서처럼 상세한 조종은 할수 없다.*

Microsoft의 독자적인 함수들의 사용방법

C/C++서고의 스레드와 관련한 함수들의 사용방법을 보기 위해 앞의 프로그람을 Microsoft 독자의 _beginthreadex( )함수를 사용하도록 변경시켜보자. CreateThread( ) 및 _beginthreadex( )의 파라메터의 의미와 배치순서는 같으므로 아래와 같은 세가지 변경만을 진행한다.

- 프로그람에 PROCESS.H 를 포함시킨다.
- 스레드함수의 선언을 _beginthreadex( )에서 사용되는 자료형의 이름으로 변경한다.
- CreateThread( )의 호출을 _beginthreadex( )의 호출로 변경한다.

프로그람에는 다중스레드대응의 서고를 련결하여야 한다.(그러자면 [Project setting] 특성표를 열고 [C/C++]표쪽을 선택하고 [Category]목록칸에서 [Code generation]을 선택하고 [사용하는 실시간서고] 의 목록칸에서 [Multi Thread]를 선택한다.) 변경을 진행한 프로그람을 실례 15-2에 보여 주었다.

실례 15-2. BeginThread 프로그람

```
// Microsoft 독자의 _beginthreadex( )함수의 사용

#include <windows.h>
#include <cstring>
#include <cstdio>
#include <process.h>
#include "thread.h"

#define MAX 5000

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);

// _beginthreadex( )에 요구되는 선언을 사용한다.
unsigned __stdcall MyThread1(void * param);
unsigned __stdcall MyThread2(void * param);

char szWinName[] = "MyWin"; // 창문클라스의 이름

char str[255]; // 표시할 문자렬을 보관한다.

DWORD Tid1, Tid2; // 스레드 ID

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
```

```
HWND hwnd;
MSG msg;
WNDCLASSEX wcl;
HACCEL hAccel;

// 창문클라스를 정의한다.
wcl.cbSize = sizeof(WNDCLASSEX);

wcl.hInstance = hThisInst;      // 실체의 손잡이
wcl.lpszClassName = szWinName; // 창문클라스의 이름
wcl.lpfnWndProc = WindowFunc;  // 창문함수
wcl.style = 0;                 // 체계설정의 형식

wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

wcl.lpszMenuName = "ThreadMenu"; // 기본차림표

wcl.cbClsExtra = 0; // 보조기억기령역은 필요 없다.
wcl.cbWndExtra = 0;

// 창문의 배경색을 흰색으로 한다.
wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

// 창문클라스를 등록한다.
if(!RegisterClassEx(&wcl)) return 0;

/* 창문클라스가 등록되였으므로
   창문을 작성할수 있다. */
hwnd = CreateWindow(
  szWinName, // 창문클라스의 이름
  "Demonstrate Threads", // 제목
  WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
  CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 너비는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 높이는 Windows가 결정하게 한다.
```

```
    NULL,          // 어미창문은 없다.
    NULL,          // 클라스차림표의 덧쓰기는 하지 않는다.
    hThisInst,     // 실체의 손잡이
    NULL           // 추가파라메터는 없다.
  );

  // 건반가속기를 적재한다.
  hAccel = LoadAccelerators(hThisInst, "ThreadMenu");

  // 창문을 표시한다.
  ShowWindow(hwnd, nWinMode);
  UpdateWindow(hwnd);

  // 통보문순환고리를 작성한다.
  while(GetMessage(&msg, NULL, 0, 0))
  {
    if(!TranslateAccelerator(hwnd, hAccel, &msg)) {
      TranslateMessage(&msg); // 건반통보를 변환한다.
      DispatchMessage(&msg); // Windows 2000에 조종을 넘긴다.
    }
  }

  return msg.wParam;
}

/* 이 함수는 Windows 2000으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받아 들인다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  int response;

  switch(message) {
    case WM_COMMAND:
      switch(LOWORD(wParam)) {
        case IDM_THREAD:
          // _beginthreadex( )를 사용하여 스레드를 작성한다.
```

```
            _beginthreadex(NULL, 0, MyThread1,
                           (LPVOID) hwnd, 0,
                           (unsigned *) &Tid1);
            _beginthreadex(NULL, 0, MyThread2,
                           (LPVOID) hwnd, 0,
                           (unsigned *) &Tid2);
            break;
          case IDM_EXIT:
            response = MessageBox(hwnd, "Quit the Program?",
                                  "Exit", MB_YESNO);
            if(response == IDYES) PostQuitMessage(0);
            break;
          case IDM_HELP:
            MessageBox(hwnd,
                       "F1: Help\nF2: Demonstrate Threads",
                       "Help", MB_OK);
            break;
        }
        break;
      case WM_DESTROY: // 프로그램을 끝낸다.
        PostQuitMessage(0);
        break;
      default:
        /* 이 switch 문에서 지정된것 이외의 통보문은
           Windows 2000 에 처리를 맡긴다. */
        return DefWindowProc(hwnd, message, wParam, lParam);
    }

    return 0;
}

// 프로세스안에서 실행되는 스레드
unsigned __stdcall MyThread1(void * param)
{
    int i;
    HDC hdc;

    for(i=0; i<MAX; i++) {
```

```
      sprintf(str, "Thread 1: loop # %5d ", i);
      hdc = GetDC((HWND) param);
      TextOut(hdc, 1, 1, str, strlen(str));
      ReleaseDC((HWND) param, hdc);
    }

    return 0;
  }

  // 프로세스안에서 실행되는 또 하나의 스레드
  unsigned __stdcall MyThread2(void * param)
  {
    int i;
    HDC hdc;

    for(i=0; i<MAX; i++) {
      sprintf(str, "Thread 2: loop # %5d ", i);
      hdc = GetDC((HWND) param);
      TextOut(hdc, 1, 20, str, strlen(str));
      ReleaseDC((HWND) param, hdc);
    }

    return 0;
  }
```

C 언어의 표준서고함수들을 대신하는 Win32 함수들

많은 다중스레드프로그람은 C/C++의 표준서고함수를 사용하지 않고도 작성할수 있다. 기억기읽기를 발생시킴이 없이 CreateThread( )와 ExitThread( )를 안심하고 사용할수 있다.

실례로 이 책에서 작성하는 많은 실례프로그람들에서는 C 언어의 표준서고함수를 *sprintf( )* 와 strlen( ) 두가지밖에 사용하지 않고 있다. Win32 에는 이 두 함수를 대신하여 쓰이는 *wsprintf( )*와 *lstrlen( )*라는 함수가 있다. Win32 의 함수에는 Unicode 에 대응한 확장기능 등이 있으나 기본적인 기능은 C/C++ 표준서고함수와 동등하다.

Win32 에는 이밖에도 lstrcat( ), lstrcmp( ) 및 lstrcat( ) 등 C/C++의 문자렬조작함수를 대신하여 쓰이는 함수가 몇개 있다. CharUpper( ), CharLower( ), IsCharAlpha( ) 및 IsCharAlphNumeric( ) 등과 같은 문자조작용함수들도 많이 있다. 만일 C/C++의 표준서고함수를 사용하여 간단한 문자조작만을 진행하고 있으면 그 함수

들 대신에 Win32 의 함수를 사용할수도 있다.

앞으로 이 장에서 작성하는 실례프로그람에서는 CreateThread( )를 사용하여 스레드를 작성한다. C/C++ 표준서고함수를 사용하지 않도록 하기 위해 sprintf( )와 strlen( )대신에 wsprintf( )와 lstrlen( )를 사용하기로 한다. 이렇게 하면 앞으로 작성하는 실례프로그람은 Windows 2000 프로그람의 작성기능을 가진 임의의 C/C++번역프로그람에서도 정확히 번역되고 실행할수 있게 된다.

스레드의 정지와 재개

*SuspendThread( )*를 사용하면 실행중의 스레드를 정지시킬수 있다. *ResumeThread( )*를 사용하면 정지중의 스레드를 재개할수 있다. 이 함수들의 선언은 다음과 같다.

```
DWORD SuspendThread(HANDLE hThread);
DWORD ResumeThread(HANDLE hThread);
```

어느 함수에서나 hThread 에는 스레드의 손잡이를 설정한다.

스레드는 *정지계수기*라는 값을 가지고 있다. 이 계수기가 령인 경우 스레드는 정지하고 있지 않으며 령이 아닌 경우는 정지된 상태로 되여 있다.

SuspendThread( )를 호출하면 정지계수기가 증가된다. ResumeThread( )를 호출하면 정지계수기가 감소된다. 정지중의 스레드는 그의 정지계수기가 령으로 되여 있을때만 재개된다. 그러므로 정지중의 스레드를 재개하기 위해서는 SuspendThread( )를 호출한 회수와 같은 회수만큼 ResumeThread( )를 호출하여야 한다.

이 두개의 함수들은 스레드의 직전의 정지계수기를 돌려 주며 오유가 발생한 경우는 -1 을 돌려 준다.

스레드의 우선권순위

매개 스레드에는 우선권순위(Priority)가 설정되여 있다. 스레드의 우선권순위는 스레드가 얻게 되는 CPU 시간을 결정한다. 우선권순위가 낮은 스레드가 얻게 되는 CPU 시간은 적고 순위가 높은 스레드가 얻는 CPU 시간은 많게 된다. 물론 스레드가 얻는 CPU 시간은 그 스레드자체의 동작만이 아니라 체계에서 동시에 실행되고 있는 다른 스레드의 동작에도 큰 영향을 미치게 된다.

스레드의 우선권순위설정은 두개 값을 조합하여 실시된다. 즉 프로세스전체의 우선권순위클라스와 그 클라스에 부속된 매개 스레드의 우선권순위이다. 다시말하여 스레드의 우선권순위는 프로세스의 *우선권순위클라스*와 매개 스레드의 우선권순위를 조합하여 결정된다. 이 두가지 우선권순위의 의미에 대해서는 다음에 설명한다.

우선권순위클라스

GetPriorityClass( )를 사용하면 현재의 우선권순위클라스를 얻을수 있다. 그리고 SetPriorityClass( )을 사용하면 우선권순위클라스를 설정할수 있다. 이 함수들의 선언은 다음과 같다.

```
DWORD GetPriorityClass(HANDLE hApp);
BOOL SetPriorityClass(HANDLE hApp, DWORD dwPriority);
```

hApp 에는 프로세스의 손잡이를 설정한다. GetPriorityClass( )는 응용프로그람의 우선권순위클라스를 돌려 주며 호출이 실패한 경우에는 령을 돌려 준다. SetPriorityClass( )에서 dwPriority 에는 프로세스의 새로운 우선권순위클라스를 설정한다. 우선권순위클라스의 값에는 다음과 같은것들이 있다. 여기에서는 우선권순위가 높은 순서로 보여 주었다.

REAL_TIME_PRIORITY_CLASS
HIGH_PRIORITY_CLASS
ABOVE_NORMAL_PRIORITY_CLASS(Windows 2000 에 추가된것)
NORMAL_PRIORITY_CLASS
BELOW_NORMAL_PRIORITY_CLASS(Windows 2000 에 추가된것)
IDLE_PRIORITY_CLASS

프로그람에는 체계설정으로 *NORMAL_PRIORITY_CLASS* 가 부여된다. 보통은 프로그람의 우선권순위클라스를 변경할 필요가 없다. 실제적인 문제로서 프로세스의 우선권순위클라스를 변경하는것으로 인하여 콤퓨터체계전체의 성능에 나쁜 영향을 주는 경우들이 있다.

례하면 프로그람의 우선권순위클라스를 REAL_TIME_PRIORITY_CLASS 로 승격시키게 되면 그 프로그람이 CPU 시간을 독점해 버리고 만다. 특수한 목적을 가지는 응용프로그람이라면 우선권순위클라스를 승격시킬 필요가 있을지도 모르지만 일반적인 응용프로그람에서는 그렇게 할 필요가 없다. 이 장에서는 프로세스의 우선권순위클라스를 체계설정값대로 한다.

Windows 2000 의 새로운기능: Windows 2000 에는 ABOVE_NORMAL_PRIORITY_CLASS 및 BELOW_NORMAL_PRIORITY_CLASS 라는 두개의 새로운 우선권순위클라스가 추가되였다. 이 우선권순위클라스에 의해 Windows 2000 에서는 우선권순위를 이전보다 더 세밀하게 조종할수 있게 되였다.

스레드의 우선권순위

어느 우선권순위클라스라고 해도 매개 스레드의 우선권순위는 프로세스내에서 스레드가 얻게 되는 CPU 시간을 결정하는것으로 된다. 스레드가 작성되면 표준적인 우선권순위가 부여된다. 그러나 스레드의 우선권순위는 그것이 실행중이라고 해도 변경시킬수 있다.

*GetThreadPriority( )*를 사용하면 스레드의 우선권순위를 얻을수 있다. *SetThreadPriority( )*를 사용하면 스레드의 우선권순위를 높이거나 낮출수 있다. 이 함수들의 선언은 다음과 같다.

```
BOOL SetThreadPriority(HANDLE hThread, int Priority);
int GetThreadPriority(HANDLE hThread);
```

두 함수에서 hThread 에는 스레드의 손잡이를 설정한다. *SetThreadPriority( )*에서는 Priority 에 새로운 우선권순위를 설정한다. *GetThreadPriority( )*는 현재의 우선권순위를 돌려 준다. 우선권순위는 다음의 값으로 표시된다. 여기에서는 우선권순위가 높은 순서로 보여 주었다.

스레드의 우선권순위	값
THREAD_PRIORITY_TIME_CRITICAL	15
THREAD_PRIORITY_HIGHEST	2
THREAD_PRIORITY_ABOVE_NORMAL	1
THREAD_PRIORITY_NORMAL	0
THREAD_PRIORITY_BELOW_NORMAL	-1
THREAD_PRIORITY_LOWEST	-2
THREAD_PRIORITY_IDLE	-15

이 값들은 프로세스의 우선권순위클라스의 범위내에서 우선권의 준위를 가리키는 값들이다. Windows 2000 에서는 프로세스의 우선권순위클라스와 스레드의 우선권순위를 조합하여 응용프로그람에 각이한 우선권순위를 설정할수 있다.

GetThreadPriority( )는 오유가 발생한 경우에 *THREAD_PRIORITY_ERROR_RETURN*을 돌려 준다.

대부분의 경우 일반적인 우선권순위클라스의 스레드라면 스레드의 우선권순위를 변경시켜도 체계전체의 동작에 영향을 미치는 일은 없다. 다음 절에서 작성하는 스레드조종판을 사용하면 프로세스내에서 스레드의 우선권순위설정을 변경시킬수 있다.(그러나 스레드의 우선권순위클라스설정은 변경시킬수 없다.)

스레드조종판의 작성

다중스레드프로그람을 작성할 때는 여러가지 우선권순위를 실지 시험해 보는것이 좋다. 또한 스레드를 정지, 재개 또는 완료하는것도 시험해 볼수 있다. 이러한것들은 지금까지 설명해온 함수들을 리용하면 간단히 실현할수 있다. 그러므로 스레드에 다양한 설정을 진행하는 스레드조종판(Thread control panel)을 작성해 보기로 한다. 스레드조종판은 다중스레드프로그람의 실행중에 사용하며 그것을 사용하면 스레드의 상태를 변경시키고 그 결과를 확인할수 있다.

이 장에서 작성하는 스레드조종판은 두개의 스레드를 조종할수 있다. 프로그람을 간단히 하기 위해 프로그람의 기본스레드의 일부로서 실행되는 양식화대화칸으로서 스레드조종판을 작성한다. 여기서는 스레드조종판이 개개의 스레드를 관리하기 위하여 대역적인 스레드손잡이를 사용한다.

스레드조종판에는 다음의 기능들이 있다.

- 스레드의 우선권순위를 설정한다.
- 스레드를 정지한다.
- 스레드를 재개한다.
- 스레드를 완료한다.

스레드조종판에는 매개 스레드의 현재의 우선권순위를 표시하는 기능도 있다.

이미 설명한것처럼 스레드조종판은 양식화대화칸으로서 작성한다. 양식화대화칸이 표시될 때는 사용자가 대화칸을 닫을 때까지 응용프로그람의 다른 부분이 정지된 상태로 된다는것을 상기해 볼 필요가 있다.

그러나 다중스레드프로그람에서는 양식화대화칸자체를 하나의 스레드로서 실행할수 있다. 이 경우에는 프로그람내의 다른 스레드를 동시에 실행하게 할수 있다. 스레드조종판은 프로그람의 기본스레드의 일부로서 실행된다. 그러므로 그자체가 하나의 스레드로 된다.

이 수법의 우점은 비양식화대화칸보다 간단히 작성할수 있는 양식화대화칸을 리용해서 비양식화대화칸과 같은 기능을 실현할수 있다는것이다. 여기서는 대화칸이 하나의 스레드로서 작성되므로 비양식화대화칸을 사용할 필요가 없다. 이러한 경험들을 축적해나

가느라면 다중스레드프로그람의 작성이 지금까지 곤난한것으로 생각되던 문제를 해결할 수 있는 유효한 수단으로 된다는것을 알수 있을것이다.

스레드조종판의 프로그람코드

실례 15-3 에 스레드조종판의 사용방법에 대한 실례를 주는 프로그람을 보여 주었다. 이 프로그람은 앞에서 작성한 스레드 실례프로그람에 대화칸을 추가한것이다. 프로그람의 실행결과는 그림 15-2 와 같다.

이 프로그람을 사용하려면 먼저 [Thread]차림표에서 [Start Thread]를 선택하여 스레드의 실행을 개시하고 스레드조종판을 표시하여야 한다. 스레드조종판이 표시되면 각이한 우선순위를 설정하거나 스레드를 정지하거나 재개하여 그 결과를 확인할수 있다.

실례 15-3. Panel 프로그람

```
// 스레드조종판의 리용

#include <windows.h>
#include "panel.h"

#define MAX 50000

#define NUMPRIORITIES 5
#define OFFSET 2

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK ThreadPanel(HWND, UINT, WPARAM, LPARAM);

DWORD WINAPI MyThread1(LPVOID param);
DWORD WINAPI MyThread2(LPVOID param);

char szWinName[] = "MyWin"; // 창문클라스의 이름

char str[255]; // 표시할 문자렬을 보관한다.

DWORD Tid1, Tid2; // 스레드ID
HANDLE hThread1, hThread2; // 스레드의 손잡이

int ThPriority1, ThPriority2;    // 스레드의 우선권순위
int suspend1 = 0, suspend2 = 0; // 스레드의 상태
```

```
char priorities[NUMPRIORITIES][80] = {
  "Lowest",
  "Below Normal",
  "Normal",
  "Above Normal",
  "Highest"
};

HINSTANCE hInst;

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;
  HACCEL hAccel;

  // 창문클라스를 정의한다.
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst;      // 실체의 손잡이
  wcl.lpszClassName = szWinName; // 창문클라스의 이름
  wcl.lpfnWndProc = WindowFunc;  // 창문함수
  wcl.style = 0;                 // 체계설정의 형식

  wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
  wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
  wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

  wcl.lpszMenuName = "ThreadPanelMenu"; // 기본차림표

  wcl.cbClsExtra = 0; // 보조기억기렴역은 필요 없다.
  wcl.cbWndExtra = 0;

  // 창문의 배경색을 흰색으로 한다.
  wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
```

```
// 창문클라스를 등록한다.
if(!RegisterClassEx(&wcl)) return 0;

/* 창문클라스가 등록되였으므로
   창문을 작성할수 있다. */
hwnd = CreateWindow(
  szWinName, // 창문클라스의 이름
  "Using a Thread Control Panel", // 제목
  WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
  CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 너비는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 높이는 Windows가 결정하게 한다.
  NULL,          // 어미창문은 없다.
  NULL,          // 클라스차림표의 덧쓰기는 하지 않는다.
  hThisInst,     // 실체의 손잡이
  NULL           // 추가파라메터는 없다.
);

hInst = hThisInst; // 실체의 손잡이를 보관한다.

// 건반가속기를 적재한다.
hAccel = LoadAccelerators(hThisInst, "ThreadPanelMenu");

// 창문을 표시한다.
ShowWindow(hwnd, nWinMode);
UpdateWindow(hwnd);

// 통보문순환고리를 작성한다.
while(GetMessage(&msg, NULL, 0, 0))
{
  if(!TranslateAccelerator(hwnd, hAccel, &msg)) {
    TranslateMessage(&msg); // 건반통보를 변환한다.
    DispatchMessage(&msg); // Windows 2000에 조종을 넘긴다.
  }
}
```

```
  return msg.wParam;
}

/* 이 함수는 Windows 2000으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받아 들인다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  int response;

  switch(message) {
    case WM_COMMAND:
      switch(LOWORD(wParam)) {
        case IDM_THREAD: // 스레드를 작성한다.
          suspend1 = suspend2 = 0;
          hThread1 = CreateThread(NULL, 0,
                            (LPTHREAD_START_ROUTINE)MyThread1,
                            (LPVOID) hwnd, 0, &Tid1);
          hThread2 = CreateThread(NULL, 0,
                            (LPTHREAD_START_ROUTINE)MyThread2,
                            (LPVOID) hwnd, 0, &Tid2);
          break;
        case IDM_PANEL: // 스레드조종판을 표시한다.
          DialogBox(hInst, "ThreadPanelDB", hwnd,
                            (DLGPROC) ThreadPanel);
          break;
        case IDM_EXIT:
          response = MessageBox(hwnd, "Quit the Program?",
                                "Exit", MB_YESNO);
          if(response == IDYES) PostQuitMessage(0);
          break;
        case IDM_HELP:
          MessageBox(hwnd,
                     "F1: Help\nF2: Start Threads\nF3: Panel",
                     "Help", MB_OK);
          break;
      }
```

```
        break;
      case WM_DESTROY: // 프로그람을 끝낸다.
        PostQuitMessage(0);
        break;
      default:
        /* 이 switch 문에서 지정된것 이외의 통보문은
           Windows 2000 에 처리를 맡긴다. */
       return DefWindowProc(hwnd, message, wParam, lParam);
    }

    return 0;
}

// 프로세스안에서 실행되는 스레드
DWORD WINAPI MyThread1(LPVOID param)
{
    int i;
    HDC hdc;

    for(i=0; i<MAX; i++) {
      wsprintf(str, "Thread 1: loop # %5d ", i);
      hdc = GetDC((HWND) param);
      TextOut(hdc, 1, 1, str, lstrlen(str));
      ReleaseDC((HWND) param, hdc);
    }

    return 0;
}

// 프로세스안에서 실행되는 다른 하나의 스레드
DWORD WINAPI MyThread2(LPVOID param)
{
    int i;
    HDC hdc;

    for(i=0; i<MAX; i++) {
      wsprintf(str, "Thread 2: loop # %5d ", i);
      hdc = GetDC((HWND) param);
```

```
    TextOut(hdc, 1, 20, str, lstrlen(str));
    ReleaseDC((HWND) param, hdc);
  }

  return 0;
}

// 스레드조종판의 대화칸
LRESULT CALLBACK ThreadPanel(HWND hdwnd, UINT message,
                             WPARAM wParam, LPARAM lParam)
{
  long i;
  HWND hpbRes, hpbSus;

  switch(message) {
    case WM_INITDIALOG:
      // 목록칸을 초기화한다.
      for(i=0; i<NUMPRIORITIES; i++) {
        SendDlgItemMessage(hdwnd, IDD_LB1,
            LB_ADDSTRING, 0, (LPARAM) priorities[i]);
        SendDlgItemMessage(hdwnd, IDD_LB2,
            LB_ADDSTRING, 0, (LPARAM) priorities[i]);
      }

      // 현재의 우선권순위를 얻는다.
      ThPriority1 = GetThreadPriority(hThread1) + OFFSET;
      ThPriority2 = GetThreadPriority(hThread2) + OFFSET;

      // 목록칸을 갱신한다.
      SendDlgItemMessage(hdwnd, IDD_LB1, LB_SETCURSEL,
                         (WPARAM) ThPriority1, 0);
      SendDlgItemMessage(hdwnd, IDD_LB2, LB_SETCURSEL,
                         (WPARAM) ThPriority2, 0);

      // 첫번째 스레드용의 [Suspend 1] 및 [Resume 1]단추를 설정한다.
      hpbSus = GetDlgItem(hdwnd, IDD_SUSPEND1);
      hpbRes = GetDlgItem(hdwnd, IDD_RESUME1);
      if(suspend1) {
```

```
        EnableWindow(hpbSus, 0); // [Suspend 1] 단추를 무효로 한다.
        EnableWindow(hpbRes, 1); // [Resume 1] 단추를 유효로 한다.
      }
      else {
        EnableWindow(hpbSus, 1); // [Suspend 1] 단추를 유효로 한다.
        EnableWindow(hpbRes, 0); // [Resume 1] 단추를 무효로 한다.
      }

      // 두번째 스레드용의 [Suspend 2] 및 [Resume 2] 단추를 설정한다.
      hpbSus = GetDlgItem(hdwnd, IDD_SUSPEND2);
      hpbRes = GetDlgItem(hdwnd, IDD_RESUME2);
      if(suspend2) {
        EnableWindow(hpbSus, 0); // [Suspend 2] 단추를 무효로 한다.
        EnableWindow(hpbRes, 1); // [Resume 2] 단추를 유효로 한다.
      }
      else {
        EnableWindow(hpbSus, 1); // [Suspend 2] 단추를 유효로 한다.
        EnableWindow(hpbRes, 0); // [Resume 2] 단추를 무효로 한다.
      }

      return 1;
    case WM_COMMAND:
      switch(wParam) {
        case IDD_TERMINATE1:
          TerminateThread(hThread1, 0);
          return 1;
        case IDD_TERMINATE2:
          TerminateThread(hThread2, 0);
          return 1;
        case IDD_SUSPEND1:
          SuspendThread(hThread1);
          hpbSus = GetDlgItem(hdwnd, IDD_SUSPEND1);
          hpbRes = GetDlgItem(hdwnd, IDD_RESUME1);
          EnableWindow(hpbSus, 0); // [Suspend 1] 단추를 무효로 한다.
          EnableWindow(hpbRes, 1); // [Resume 1] 단추를 유효로 한다.
          suspend1 = 1;
          return 1;
        case IDD_RESUME1:
```

```
            ResumeThread(hThread1);
            hpbSus = GetDlgItem(hdwnd, IDD_SUSPEND1);
            hpbRes = GetDlgItem(hdwnd, IDD_RESUME1);
            EnableWindow(hpbSus, 1); // [Suspend 1] 단추를 유효로 한다.
            EnableWindow(hpbRes, 0); // [Resume 1] 단추를 무효로 한다.
            suspend1 = 0;
            return 1;
          case IDD_SUSPEND2:
            SuspendThread(hThread2);
            hpbSus = GetDlgItem(hdwnd, IDD_SUSPEND2);
            hpbRes = GetDlgItem(hdwnd, IDD_RESUME2);
            EnableWindow(hpbSus, 0); // [Suspend 2] 단추를 무효로 한다.
            EnableWindow(hpbRes, 1); // [Resume 2] 단추를 유효로 한다.
            suspend2 = 1;
            return 1;
          case IDD_RESUME2:
            ResumeThread(hThread2);
            hpbSus = GetDlgItem(hdwnd, IDD_SUSPEND2);
            hpbRes = GetDlgItem(hdwnd, IDD_RESUME2);
            EnableWindow(hpbSus, 1); // [Suspend 2] 단추를 유효로 한다.
            EnableWindow(hpbRes, 0); // [Resume 2] 단추를 무효로 한다.
            suspend2 = 0;
            return 1;
          case IDOK: // 우선권순위를 변경한다.
            ThPriority1 = SendDlgItemMessage(hdwnd, IDD_LB1,
                                   LB_GETCURSEL, 0, 0);
            ThPriority2 = SendDlgItemMessage(hdwnd, IDD_LB2,
                                   LB_GETCURSEL, 0, 0);
            SetThreadPriority(hThread1, ThPriority1-OFFSET);
            SetThreadPriority(hThread2, ThPriority2-OFFSET);
            return 1;
          case IDCANCEL:
            EndDialog(hdwnd, 0);
            return 1;
        }
    }
    return 0;
}
```

이 프로그람은 아래와 같은 내용의 PANEL.H 라는 머리부파일을 사용한다.

```
#define IDM_THREAD          100
#define IDM_HELP            101
#define IDM_PANEL           102
#define IDM_EXIT            103

#define IDD_LB1             200
#define IDD_LB2             201
#define IDD_TERMINATE1      202
#define IDD_TERMINATE2      203
#define IDD_SUSPEND1        204
#define IDD_SUSPEND2        205
#define IDD_RESUME1         206
#define IDD_RESUME2         207
#define IDD_TEXT1           208
#define IDD_TEXT2           209
#define IDD_TEXT3           210
```

이 프로그람에서 사용되는 자원파일의 내용을 아래에 보여 주었다.

```
#include <windows.h>
#include "panel.h"

ThreadPanelMenu MENU
{
  POPUP "&Threads" {
    MENUITEM "&Start Threads\tF2", IDM_THREAD
    MENUITEM "&Control Panel\tF3", IDM_PANEL
    MENUITEM "E&xit\tCtrl+X", IDM_EXIT
  }
  MENUITEM "&Help", IDM_HELP
}

ThreadPanelDB DIALOGEX 20, 20, 170, 140
CAPTION "Thread Control Panel"
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
{
```

```
  DEFPUSHBUTTON "Change", IDOK, 80, 105, 33, 14
  PUSHBUTTON "Done", IDCANCEL, 15, 120, 33, 14
  PUSHBUTTON "Terminate 1", IDD_TERMINATE1, 10, 10, 42, 12
  PUSHBUTTON "Terminate 2", IDD_TERMINATE2, 10, 60, 42, 12
  PUSHBUTTON "Suspend 1", IDD_SUSPEND1, 10, 25, 42, 12
  PUSHBUTTON "Resume 1", IDD_RESUME1, 10, 40, 42, 12
  PUSHBUTTON "Suspend 2", IDD_SUSPEND2, 10, 75, 42, 12
  PUSHBUTTON "Resume 2", IDD_RESUME2, 10, 90, 42, 12
  LISTBOX IDD_LB1, 65, 11, 63, 42, LBS_NOTIFY | WS_VISIBLE |
          WS_BORDER | WS_VSCROLL | WS_TABSTOP
  LISTBOX IDD_LB2, 65, 61, 63, 42, LBS_NOTIFY | WS_VISIBLE |
          WS_BORDER | WS_VSCROLL | WS_TABSTOP
  CTEXT "Thread 1", IDD_TEXT1, 140, 22, 24, 18
  CTEXT "Thread 2", IDD_TEXT2, 140, 73, 24, 18
  CTEXT "Thread Priority", IDD_TEXT3, 65, 0, 64, 10
}

ThreadPanelMenu ACCELERATORS
{
  VK_F1, IDM_HELP, VIRTKEY
  VK_F2, IDM_THREAD, VIRTKEY
  VK_F3, IDM_PANEL, VIRTKEY
  "^X", IDM_EXIT
}
```

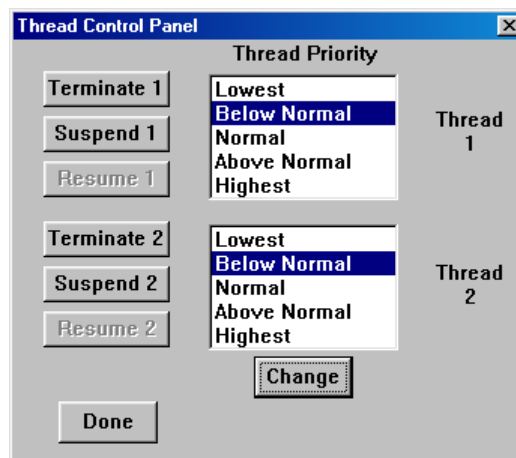



그림 15-2. 스레드조종판의 실행결과

스레드조종판의 상세

스레드조종판의 내용을 자세히 살펴 보자. 프로그람의 선두에서는 스레드조종판에서 리용되는 몇개의 대역변수가 선언되여 있다.

```
DWORD Tid1, Tid2; // 스레드ID
HANDLE hThread1, hThread2; // 스레드손잡이

int ThPriority1, ThPriority2;   // 스레드의 우선권순위
int suspend1 = 0, suspend2 = 0; // 스레드의 상태

char priorities[NUMPRIORITIES][80] = {
  "Lowest",
  "Below Normal",
  "Normal",
  "Above Normal",
  "Highest"
};
```

Tid1과 Tid2는 두 스레드의 ID를 보관한다. hThread1과 hThread2는 두 스레드의 손잡이를 보관한다. 이 손잡이에는 CreateThread()로 스레드를 작성했을 때의 돌림값이 보관된다. ThPriority1과 ThPriority2는 두 스레드의 현재의 우선권순위를 보관한다. suspend1과 suspend2는 두 스레드의 상태를 보관하는데 사용된다. 배렬 priorities는 스레드조종판에서 사용되는 목록칸을 초기화하는 문자렬을 보관하고 있다. 이 문자렬은 우선권순위를 보여 주는것들이다.

프로그람에는 다음의 마크로들도 정의되여 있다.

```
#define NUMPRIORITIES      5
#define OFFSET             2
```

NUMPRIORITIES 는 설정가능한 우선권순위의 개수를 가리키는 마크로이다. 스레드조종판을 사용하여 설정할수 있는 스레드의 우선권순위는 다음과 같다.

THREAD_PRIORITY_HIGHEST
THREAD_PRIORITY_ABOVE_NORMAL
THREAD_PRIORITY_NORMAL
THREAD_PRIORITY_BELOW_NORMAL
THREAD_PRIORITY_LOWEST

다음의 두개의 우선권순위는 설정할수 없다. 왜냐하면 이 두개의 우선권순위가 스레드조종판에서는 거의나 의미를 가지지 않기때문이다. 례를 들어 만일 스레드의 우선권순위에 THREAD_PRIORITY_TIME_CRITICAL 을 설정하는 응용프로그람을 작성한다면 우선권순위클라스에 *REALTIME_PRIORITY_CLASS* 을 설정하는 편이 더 유익하기때문이다.

THREAD_PRIORITY_TIME_CRITICAL
THREAD_PRIORITY_IDLE

OFFSET 는 목록칸의 색인값을 스레드의 우선권순위의 값으로 변환하기 위한것이다. *THREAD_PRIORITY_NORMAL* 의 값이 령이라는것을 기억하고 있을것이다. 이 프로그람에서는 가장 높은 우선권순위가 THREAD_PRIORITY_HIGHEST(값은 2)이며 제일 낮은 우선권순위가 THREAD_PRIORITY_LOWEST(값은 –2)로 되여 있다. 목록칸의 색인은 령으로부터 시작되므로 색인값에서 OFFSET(값은 2)를 덜면 우선권순위값으로 변환할수 있다.

스레드조종판의 대화함수는 다음과 같다.

```
// 스레드조종판의 대화칸
LRESULT CALLBACK ThreadPanel(HWND hdwnd, UINT message,
                             WPARAM wParam, LPARAM lParam)
{
  long i;
  HWND hpbRes, hpbSus;

  switch(message) {
    case WM_INITDIALOG:
      // 목록칸을 초기화한다.
      for(i=0; i<NUMPRIORITIES; i++) {
        SendDlgItemMessage(hdwnd, IDD_LB1,
            LB_ADDSTRING, 0, (LPARAM) priorities[i]);
        SendDlgItemMessage(hdwnd, IDD_LB2,
            LB_ADDSTRING, 0, (LPARAM) priorities[i]);
       }

       // 현재의 우선권순위를 얻는다.
       ThPriority1 = GetThreadPriority(hThread1) + OFFSET;
       ThPriority2 = GetThreadPriority(hThread2) + OFFSET;
```

```
      // 목록칸을 갱신한다.
      SendDlgItemMessage(hdwnd, IDD_LB1, LB_SETCURSEL,
                         (WPARAM) ThPriority1, 0);
      SendDlgItemMessage(hdwnd, IDD_LB2, LB_SETCURSEL,
                         (WPARAM) ThPriority2, 0);

      // 첫번째 스레드용의 [Suspend 1] 및 [Resume 1]단추를 설정한다.
      hpbSus = GetDlgItem(hdwnd, IDD_SUSPEND1);
      hpbRes = GetDlgItem(hdwnd, IDD_RESUME1);
      if(suspend1) {
        EnableWindow(hpbSus, 0); // [Suspend 1]단추를 무효로 한다.
        EnableWindow(hpbRes, 1); // [Resume 1]단추를 유효로 한다.
      }
      else {
        EnableWindow(hpbSus, 1); // [Suspend 1]단추를 유효로 한다.
        EnableWindow(hpbRes, 0); // [Resume 1]단추를 무효로 한다.
      }

      // 두번째 스레드용의 [Suspend 2] 및 [Resume 2]단추를 설정한다.
      hpbSus = GetDlgItem(hdwnd, IDD_SUSPEND2);
      hpbRes = GetDlgItem(hdwnd, IDD_RESUME2);
      if(suspend2) {
        EnableWindow(hpbSus, 0); // [Suspend 2]단추를 무효로 한다.
        EnableWindow(hpbRes, 1); // [Resume 2]단추를 유효로 한다.
      }
      else {
        EnableWindow(hpbSus, 1); // [Suspend 2]단추를 유효로 한다.
        EnableWindow(hpbRes, 0); // [Resume 2]단추를 무효로 한다.
      }

      return 1;
   case WM_COMMAND:
     switch(wParam) {
       case IDD_TERMINATE1:
         TerminateThread(hThread1, 0);
         return 1;
       case IDD_TERMINATE2:
```

```
      TerminateThread(hThread2, 0);
      return 1;
    case IDD_SUSPEND1:
      SuspendThread(hThread1);
      hpbSus = GetDlgItem(hdwnd, IDD_SUSPEND1);
      hpbRes = GetDlgItem(hdwnd, IDD_RESUME1);
      EnableWindow(hpbSus, 0); // [Suspend 1] 단추를 무효로 한다.
      EnableWindow(hpbRes, 1); // [Resume 1] 단추를 유효로 한다.
      suspend1 = 1;
      return 1;
    case IDD_RESUME1:
      ResumeThread(hThread1);
      hpbSus = GetDlgItem(hdwnd, IDD_SUSPEND1);
      hpbRes = GetDlgItem(hdwnd, IDD_RESUME1);
      EnableWindow(hpbSus, 1); // [Suspend 1] 단추를 유효로 한다.
      EnableWindow(hpbRes, 0); // [Resume 1] 단추를 무효로 한다.
      suspend1 = 0;
      return 1;
    case IDD_SUSPEND2:
      SuspendThread(hThread2);
      hpbSus = GetDlgItem(hdwnd, IDD_SUSPEND2);
      hpbRes = GetDlgItem(hdwnd, IDD_RESUME2);
      EnableWindow(hpbSus, 0); // [Suspend 2] 단추를 무효로 한다.
      EnableWindow(hpbRes, 1); // [Resume 2] 단추를 유효로 한다.
      suspend2 = 1;
      return 1;
    case IDD_RESUME2:
      ResumeThread(hThread2);
      hpbSus = GetDlgItem(hdwnd, IDD_SUSPEND2);
      hpbRes = GetDlgItem(hdwnd, IDD_RESUME2);
      EnableWindow(hpbSus, 1); // [Suspend 2] 단추를 유효로 한다.
      EnableWindow(hpbRes, 0); // [Resume 2] 단추를 무효로 한다.
      suspend2 = 0;
      return 1;
    case IDOK: // 우선권순위를 변경한다.
      ThPriority1 = SendDlgItemMessage(hdwnd, IDD_LB1,
                          LB_GETCURSEL, 0, 0);
      ThPriority2 = SendDlgItemMessage(hdwnd, IDD_LB2,
```

```
                                    LB_GETCURSEL, 0, 0);
            SetThreadPriority(hThread1, ThPriority1-OFFSET);
            SetThreadPriority(hThread2, ThPriority2-OFFSET);
            return 1;
          case IDCANCEL:
            EndDialog(hdwnd, 0);
            return 1;
        }
    }
    return 0;
}
```

스레드조종판이 표시되면 다음의 순서로 처리가 진행된다.

① 스레드조종판에서 사용되는 두개의 목록칸을 초기화한다.
② 두 스레드의 현재의 우선권순위를 얻는다.
③ 두 스레드의 우선권순위를 목록칸에 반전표시한다.
④ 스레드가 정지된 경우는 스레드의 [Suspend]단추를 무효로 한다. 스레드가 실행중인 경우는 스레드의 [Resume]단추를 무효로 한다.

대화칸이 초기화되면 목록칸에서 새로운 우선권순위를 선택하고 [Change]단추를 눌러서 스레드의 우선권순위를 변경시킬수 있다.

[Suspend]단추를 누르면 스레드를 정지시킬수 있다. 대역변수 suspend1 및 suspend2 에는 매개 스레드의 현재상태가 보관된다. 변수의 값이 령이면 스레드가 실행중이라는것을 가리킨다. 령이 아니면 스레드가 정지상태라는것을 가리킨다.

정지중의 스레드를 재개하려면 [Resume]단추를 누른다. suspend1 과 suspend2 는 또한 대화칸이 초기화될 때 [Suspend]단추를 무효로 하는 역할도 한다. 어떤 스레드에 있어서나 그것을 재개하려면 SuspendThread( )의 호출과 같은 회수로 ResumeThread( )를 호출해야 한다는것을 잊지 말아야 한다. 정지된 스레드의 [Suspend]단추를 무효로 하여 SuspendThread( )가 다시 호출되는것을 방지할수 있다.

스레드의 실행중에도 스레드조종판을 열거나 닫을수 있으므로 대화칸이 초기화될 때마다 [Suspend] 및 [Resume]단추의 상태를 적절히 설정해야 한다. suspend1 과 suspend2 의 두 변수는 새로운 스레드가 개시될 때마다 령으로 재설정된다.

[Terminate]단추를 누르면 스레드를 정지시킬수 있다. 스레드가 정지되면 그것을 재개할수 있다. 스레드조종판에서는 스레드를 정지시키는데 TerminateThread( )을 사용하고 있는 점에 주의를 돌려야 한다. 이 장을 시작하면서 설명한것처럼 이 함수는 주의하여 사용해야 하는 함수이다. 그러나 스레드조종판을 사용하여 스레드의 조종을 시험

하는것뿐이라면 특별히 문제가 발생하지는 않을것이다.

앞으로 더 나아가기전에 스레드조종판을 사용하여 여러가지 우선권순위의 효과 등을 실제로 확인해 보는것이 유익하다.

다시 한보 전진

기본스레드의 우선권순위의 변경

스레드조종판을 사용해 보거나 그것에 대한 설명을 읽으면 한가지 의문점이 생길것이다. 그것은 스레드조종판이 왜 프로그람에서 작성된 두개의 스레드만을 조종하는가? 어째서 세개는 아닌가? 하는 의문이다.

모든 프로그람은 적어도 하나의 스레드를 가지며 그것을 기본스레드라고 부른다는것을 상기해 볼 필요가 있다. 실례프로그람에 있어서 기본스레드는 프로그람의 실행중에 동시에 동작하는 세개의 스레드로 되여 있다. 그러나 기본스레드를 스레드조종판에서 조종할수는 없다. 그 리유는 두가지이다.

첫째 리유는 보통 기본스레드의 우선권순위를 변경시킬 필요가 없다는것이다. 추가된 스레드의 우선권순위만을 변경하는것이 일반적이다. 두번째 리유는 스레드와 관련된 모든 조종을 기본스레드에 대해서 진행할수 없기때문이다. 실례로 기본스레드를 정지시키면 프로그람자체를 완료할수 없게 되여 버리고 만다. 더우기 스레드조종판으로 되여 있는 대화칸이 기본스레드의 일부이므로 만약 기본스레드를 정지시키면 그것을 재개할 수단이 없게 되여 버리고 만다.

이러한 리유들로부터 기본스레드는 체계설정(THREAD_PRIORITY_NORMAL) 그대로 실행해야 하는것이다.

그러나 기본스레드의 우선권순위를 참조하거나 변경하는것이 불가능하다는것은 아니다. 기본스레드의 손잡이를 얻으면 그것이 가능하다. *GetCurrentThread( )*를 사용하면 기본스레드의 손잡이를 얻을수 있다. 선언은 다음과 같다.

```
HANDLE GetCurrentThread(void);
```

GetCurrentThread( )는 현재스레드의 의사손잡이를 돌려 준다. 이 의사손잡이를 일반적인 손잡이가 설정되는 임의의 장면에서 사용할수 있다.

기본스레드의 우선권순위를 변경시킨 효과를 확인하려면 스레드조종판프로그람에서 IDM_THREAD 의 case 문부분을 다음과 같은 프로그람코드로 치환하고 hThread2 에 기본스레드의 손잡이가 설정되도록 하면 된다.

```
hThread1 = CreateThread(NULL, 0,
                (LPTHREAD_START_ROUTINE)MyThread1,
```

```
                        (LPVOID) hwnd, 0, &Tid1);
                CreateThread(NULL, 0,
                        (LPTHREAD_START_ROUTINE)MyThread2,
                        (LPVOID) hwnd, 0, &Tid2);
    //; hThread2에 기본스레드의 손잡이를 설정한다.
    hThread2 = GetCurrentThread( );
```

프로그람을 실행하고 스레드조종판을 표시하면 Thread2 는 기본스레드를 표시하게 된다. 그러나 주의해야 한다. 만약 기본스레드를 정지시켜 버리면 과제관리자를 사용하지 않고서는 프로그람을 완료할수 없게 된다.

동 기 화

다중스레드나 다중프로세스를 사용하는 경우에는 그것들의 동작을 협조해야 할 필요가 제기되는 경우가 있다. 이것을 *동기화*라고 한다. 동기화가 리용되는 가장 일반적인 장면은 두개이상의 스레드가 공유자원을 호출할 필요가 제기되는 경우에 그 공유자원에는 동시에 한개의 스레드밖에 호출할수 없는 때이다.

례하면 첫 스레드가 파일의 쓰기를 진행하고 있을 때는 두번째 스레드가 같은 파일을 동시에 호출하는것을 방지하여야 한다. 이 문제를 해결하기 위한 방법을 *순차화*라고 한다.

동기화가 필요되는 다른 장면으로서 한 스레드가 다른 스레드에 의해 발생되는 사건을 대기하고 있는 때가 있다. 이런 경우에는 목적하는 사건이 발생할 때까지 첫 스레드를 정지상태로 방임해 놓기 위한 어떤 수단이 필요하게 된다. 그러다가 사건이 발생하면 스레드를 재개하여야 한다.

우선 몇가지 용어를 정의해 두자. 과제에는 기본적으로 두개의 상태가 있다. 첫번째 상태는 실행상태(또는 시간구획(time slice)이 할당되면 곧 실행을 개시할수 있는 상태)이다. 두번째 상태는 차단상태이다. 이것은 필요한 자원에 대한 호출이 가능하게 되든가 혹은 목적하는 사건이 발생할 때까지 실행이 정지되여 있는 상태이다.

만약 동기화나 순차화문제와 관련한 지식이 없으면 다음 절에서 가장 일반적인 동기화의 해결책인 *신호기*(semaphore)에 대한 설명을 읽으면 된다.(충분한 지식을 소유하고 있는 사람은 이 절을 뛰여 넘어도 된다.)

동기화의 문제점과 해결책

공유자원을 호출하기 위한 순차화기능을 제공하는것은 Windows 2000 의 임무로 된다. 그것은 조작체계의 방조가 없으면 스레드나 프로세스는 자원이 다른 스레드 혹은 프

로세스로부터 호출되는가 아닌가를 알수 없기때문이다.

이 문제를 리해하기 위해 순차화를 지원하지 않는 조작체계에서 다중과제프로그람을 작성한 경우를 고찰해 보자.

동시에 실행되는 A 혹은 B라는 프로세스가 있고 그것들이 때때로 R라는 자원(디스크상의 파일 등)을 호출한다고 하자. 이 자원을 호출할수 있는것은 동시에 하나의 프로세스 또는 스레드뿐이다. 한 프로세스가 R 를 호출하고 있을 때 다른 프로세스로부터의 호출을 방지하기 위한 수단으로서 다음과 같은것을 시험해 보자.

우선 량쪽의 프로세스에서 모두 참조할수 있는 flag 라는 변수를 준비한다. 한 프로세스가 flag 를 령으로 초기화한다. 다음으로 R 를 호출하는 프로세스는 flag 가 령으로 재설정되여 있는가를 확인하고 나서 flag 를 1 로 설정하고 자원 R 를 호출한다. 마지막으로 flag 를 령으로 복귀시킨다. 이리하여 R 를 호출하는 프로그람코드는 다음과 같이 된다.

```
while(flag); // flag 가 령으로 재설정되기를 기다린다.
flag = 1; //flag 를 1 로 설정한다.

//.... 자원 R 를 호출한다.
flag = 0; // flag 를 령으로 재설정한다.
```

이 수법의 요점은 flag에 1이 설정되여 있는 경우에는 다른 프로세스가 R를 호출할수 없다는것이다. 이 수법은 정당한것처럼 생각된다. 그런데 실제로는 단순한 리유로부터 항상 정확하게 동작한다고는 단정할수 없다. 그 리유를 설명해 보자.

이 프로그람코드를 사용하면 량쪽의 프로세스가 동시에 R 를 호출하는것이 가능하게 되여 버리고 만다. while 순환에서는 flag 값의 적재와 참조가 반복되고 있다. 즉 flag 의 값이 검사되고 있다. flag 의 값이 0 으로 재설정되여 있는 경우는 프로그람의 다음 행에서 flag 의 값이 1 로 설정된다.

문제로 되는것은 이 두 처리가 다른 시간구획에서 실행된다는것이다. 두 시간구획의 사이에 다른 프로세스가 flag 의 값을 참조할수 있다. 그러므로 량쪽의 프로세스가 R 를 동시에 호출할수 있게 된다.

이 문제를 보다 상세히 설명해 보자. 프로세스 A 가 while 순환에 들어 가 flag 가 령인가를 검사한다. 다시말하여 R 의 호출이 가능하다는것을 알게 된다. 그런데 프로세스 A가 flag 를 1로 설정하기전에 그의 시간구획이 끝나면 프로세스 B의 실행이 재개된다. 만일 프로세스 B 가 while 순환을 실행하였다면 flag 가 설정되여 있지 않으므로 R 를 호출하여도 문제가 없다고 판정해 버리고 만다. 그러나 프로세스 A 의 실행이 재개되면 R 는 호출되고 만다.

이 문제의 위험성은 flag 의 검사와 설정이 다른 처리에 의해 새치기된다는것이다. 이미 설명한것처럼 flag 의 검사와 설정은 서로 다른 시간구획으로 된다. 어떠한 고안을 한다고 해도 응용프로그람의 프로그람코드를 사용하는 한에 있어서는 R 를 동시에 호출

하는것이 반드시 하나의 프로세스만이라고 확인할수 없는것이다.

동기화와 관련한 문제는 단순한것이므로 그의 해결책도 명쾌하다. 그것은 조작체계(여기에서는 Windows 2000)가 다른것에 의해 새치기되지 않는 하나의 처리안에서 기발의 참조와 설정을 진행하는 기능을 제공하면 되는것이다.

조작체계전문가들은 이 기능을 <검사 및 설정조작>이라고도 한다. 력사적인 경위로부터 순차화를 조종하는데 사용되며 스레드(또는 프로세스)간 동기화를 보장하는 기능을 제공하는 기발을 *신호기*(semaphore)라고 부른다. 신호기는 Windows 2000 이 순차화를 실현하는데서 핵심으로 된다.

Windows 2000 의 동기화객체

Windows 2000 은 다섯가지 종류의 동기화객체를 제공하고 있다.

첫번째 동기화객체는 고전적인 신호기이다. 신호기는 자원을 호출할수 있는 프로세스나 스레드의 수를 제한할수 있게 한다. 신호기를 사용하면 자원이 완전히 순차화되며 동시에 하나의 스레드 혹은 프로세스만이 호출할수 있게 된다. 그러나 신호기는 동시에 호출할수 있는 스레드나 프로세스를 한개이상의 임의의 수로 설정할수도 있다.

신호기는 계수기형식으로 실현된다. 이 계수기는 과제에 신호기가 전달되였을 때 증가되며 과제가 신호기를 해제했을 때 감소된다.

두번째 동기화객체는 *2값신호기*(Mutex semaphore)이다. 2값신호기는 자원을 순차화하는데 사용되며 동시에 호출할수 있는 스레드 혹은 프로세스를 한개만으로 제한한다. 2값신호기는 일반적인 신호기의 특수한 형식이다.

세번째 동기화객체는 *사건객체*이다. 사건객체는 다른 스레드나 프로세스로부터 자원을 사용해도 좋다는 신호가 올 때까지 자원에 대한 호출을 차단하는데 사용된다.(이것은 사건객체가 특정의 사건의 발생에 대해 통지한다는것이다.)

네번째 동기화객체는 *기다림시계*이다. 기다림시계는 특정한 시간동안 스레드의 실행을 차단한다. 기다림시계는 배경으로 실행되는 과제에서 특히 효과를 발휘한다.

다섯번째 동기화객체는 *림계구역*(Critical section)이다. 림계구역객체를 리용하여 프로그람의 부분적코드를 림계구역으로 하면 그것이 동시에 한개이상의 스레드에서 실행되는것을 방지할수 있다. 한 스레드가 림계구역의 실행을 개시하면 그것의 실행이 끝날 때까지 다른 스레드로부터 실행할수 없게 된다.(림계구역은 프로세스내의 스레드에만 적용된다.)

림계구역을 제외한 동기화객체들은 프로세스안의 스레드와 프로세스자체에 적용할수 있다. 실제로는 부분적인 통신수단으로서 가장 간단한 신호기가 잘 사용된다.

이 장에서는 신호기, 사건객체 및 기다림시계의 작성방법과 사용방법에 대해 설명한다. 이 동기화객체들을 리해하게 되면 2 값신호기나 림계구역에 대해서도 자체로 정통할수 있을것이다.

모든 동기화객체들의 핵심구조는 그것을 직접 사용하는가 혹은 내부적으로 사용하는가에 관계 없이 신호기의 개념이다. 그러므로 우선 신호기의 사용방법에 대한 설명으로

부터 시작하자.

스레드의 동기화에 신호기를 리용하는 방법

신호기를 리용하자면 그것을 *CreateSemaphore( )*를 사용하여 작성하여야 한다. 선언은 다음과 같다.

```
HANDLE CreateSemaphore(LPSECURITY_ATTRIBUTES lpSecAttr,
                       LONG InitialCount,
                       LONG MaxCount,
                       LPCSTR lpszName);
```

lpSecAttr 는 보안속성에 대한 지시자이다. lpSecAttr 에 NULL 을 설정하면 체계설정의 보안서술자가 사용된다.

신호기는 한개이상의 과제에 객체에 대한 호출을 허가한다. 객체를 동시에 호출할수 있는 과제의 수는 MaxCount 에 설정한다. MaxCount 의 값에 1 을 설정하면 신호기는 2 값신호기와 같은 기능을 가지게 되며 자원을 동시에 호출할수 있는 스레드 혹은 프로세스가 한개로 제한된다.

신호기는 현재호출을 허가하고 있는 과제의 수를 관리하기 위한 계수기를 가지고 있다. 이 계수기의 값이 령이면 과제가 신호기를 해제할 때까지 다른 호출이 금지된다. 계수기의 값이 령보다 크면 신호기에 신호가 통지된 상태이며 다른 스레드에 호출이 허가된다. 스레드에 호출이 허가될 때마다 계수기의 값이 감소된다.

신호기의 계수기의 초기값은 InitialCount 에 설정한다. 계수기의 초기값이 령이면 어떤 프로그람이 신호기를 해제할 때까지 신호기를 가지고 있는 모든 객체는 차단된다.

일반적으로 계수기의 초기값에는 1 이상의 값을 설정하며 신호기가 적어도 한개이상의 과제를 허가하는 상태로 한다. 어떠한 경우에도 InitialCount 의 값은 령이상이며 MaxCount 에 설정된 값이하여야 한다.

lpszName 에는 신호기객체의 이름을 표시하는 문자렬을 설정한다. 신호기는 몇개의 프로세스로부터 사용될수 있는 대역객체이다. 두개의 프로세스가 각각 같은 이름으로 신호기를 작성한 경우에는 두 프로세스가 같은 신호기를 참조하게 된다. 이에 의해 두 프로세스의 동기화가 달성된다.

신호기의 이름을 NULL 로 할수도 있다. 이 경우에 신호기는 한 프로세스내에서만 사용되게 된다. lpszName 에 기존의 신호기의 이름을 설정하는 경우에는 InitialCount 와 MaxCount 의 값을 설정할 필요가 없다.

*CreateSemaphore( )*함수는 호출이 성공한 경우에는 신호기의 손잡이를 돌려 주며

실패한 경우에는 NULL 을 돌려 준다. 작성한 신호기를 사용하려면 *WaitForSingleObject( )* 및 *ReleaseSemaphore( )*를 호출한다. 이 함수들의 선언은 다음과 같다.

```
DWORD WaitForSingleSemaphore(HANDLE hObject,
                                                DWORD dxHowLong);
BOOL ReleaseSemaphore(HANDLE hSema,
                                LONG Count, LPLONG lpPrevCount);
```

WaitForSingleSemaphore( )는 신호기(또는 어떤 동기화객체)를 기다리는것이다. 이 함수는 객체가 참조가능하게 되든가 림시중단(time out)상태로 될 때까지 돌림값을 돌려 주지 않는다. 신호기를 사용하는 경우는 hObject 에 기존의 신호기의 손잡이를 설정한다.

dwHowLong 파라메터에는 ㎳단위로 림시중단될 때까지의 대기시간을 설정한다. 이 시간이 경과하면 림시중단오유가 돌려 진다. 무한히 대기하게 하려면 중단시간에 INFINITE 를 설정한다.

이 함수는 호출이 성공하면 WAIT_OBJECT_0 을 돌려 준다.(이것은 호출이 허가되였다는것을 의미한다.) 림시중단된 경우는 WAIT_TIMEOUT 를 돌려 준다. WaitForSingleSemaphore( )의 호출이 성공한 경우에는 신호기의 계수기가 감소된다.

ReleaseSemaphore( )는 신호기를 해제하고 다른 스레드에 신호기의 사용을 허가한다. hSema 에는 신호기의 손잡이를 설정한다. Count 에는 신호기의 계수기에 더할 값을 설정한다. 일반적으로 이 값은 1 로 된다. lpPrevCount 파라메터에는 신호기의 계수기의 직전값이 돌려 진다. 이 계수기의 값이 필요 없으면 lpPrevCount 에 NULL 을 설정한다. 이 함수는 호출이 성공하면 령이 아닌 값을 돌려 주며 실패하면 령을 돌려 준다.

실례 15-4 에 준 프로그람은 신호기의 사용방법을 보여 주기 위한것이다. 이 프로그람은 이 장에서 처음 작성한 다중스레드프로그람을 개조하여 두개의 스레드를 동시에 실행할수 없게 하였다. 다시말하여 두개의 스레드는 순차화되여 있다.

신호기의 손잡이는 창문의 작성시에 설정되는 대역변수로 되여 있으므로 프로그람의 모든 스레드(기본스레드를 포함)에서 참조할수 있게 되여 있는 점에 주의를 돌려야 한다. 이 프로그람은 앞의 실례프로그람과 같은 머리부파일과 자원파일을 사용한다.

실례 15-4. Semaphore 프로그람

```
// 신호기를 사용하는 다중스레드프로그람


#include <windows.h>
#include "thread.h"
```

```
#define MAX 5000

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
DWORD WINAPI MyThread1(LPVOID param);
DWORD WINAPI MyThread2(LPVOID param);

char szWinName[] = "MyWin"; // 창문클라스의 이름

char str[255]; // 표시할 문자렬을 보관한다.

DWORD Tid1, Tid2; // 스레드 ID

HANDLE hSema; // 신호기의 손잡이

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;
  HACCEL hAccel;

  // 창문클라스를 정의한다.
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst;     // 실체의 손잡이
  wcl.lpszClassName = szWinName; // 창문클라스의 이름
  wcl.lpfnWndProc = WindowFunc;  // 창문함수
  wcl.style = 0;                 // 체계설정의 형식

  wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
  wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
  wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

  wcl.lpszMenuName = "ThreadMenu"; // 기본차림표

  wcl.cbClsExtra = 0; // 보조기억기령역은 필요 없다.
  wcl.cbWndExtra = 0;
```

```
// 창문의 배경색을 흰색으로 한다.
wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

// 창문클라스를 등록한다.
if(!RegisterClassEx(&wcl)) return 0;

/* 창문클라스가 등록되였으므로
   창문을 작성할수 있다. */
hwnd = CreateWindow(
  szWinName, // 창문클라스의 이름
  "Use a Semaphore", // 제목
  WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
  CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 너비는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 높이는 Windows가 결정하게 한다.
  NULL,          // 어미창문은 없다.
  NULL,          // 클라스차림표의 덧쓰기는 하지 않는다.
  hThisInst,     // 실체의 손잡이
  NULL           // 추가파라메터는 없다.
);

// 건반가속기를 적재한다.
hAccel = LoadAccelerators(hThisInst, "ThreadMenu");

// 창문을 표시한다.
ShowWindow(hwnd, nWinMode);
UpdateWindow(hwnd);

// 통보문순환고리를 작성한다.
while(GetMessage(&msg, NULL, 0, 0))
{
  if(!TranslateAccelerator(hwnd, hAccel, &msg)) {
    TranslateMessage(&msg); // 건반통보를 변환한다.
    DispatchMessage(&msg); // Windows 2000에 조종을 넘긴다.
  }
}
```

```
  return msg.wParam;
}

/* 이 함수는 Windows 2000 으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받아 들인다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  int response;

  switch(message) {
    case WM_CREATE:
      hSema = CreateSemaphore(NULL, 1, 1, NULL);
      break;
    case WM_COMMAND:
      switch(LOWORD(wParam)) {
        case IDM_THREAD:
          CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)MyThread1,
                       (LPVOID) hwnd, 0, &Tid1);
          CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)MyThread2,
                       (LPVOID) hwnd, 0, &Tid2);
          break;
        case IDM_EXIT:
          response = MessageBox(hwnd, "Quit the Program?",
                                "Exit", MB_YESNO);
          if(response == IDYES) PostQuitMessage(0);
          break;
        case IDM_HELP:
          MessageBox(hwnd,
                     "F1: Help\nF2: Demonstrate Threads",
                     "Help", MB_OK);
          break;
      }
      break;
    case WM_DESTROY: // 프로그람을 끝낸다.
      PostQuitMessage(0);
```

```
        break;
      default:
        /* 이 switch 문에서 지정된것 이외의 통보문은
           Windows 2000 에 처리를 맡긴다. */
       return DefWindowProc(hwnd, message, wParam, lParam);
    }

    return 0;
  }

  // 프로세스안에서 실행되는 스레드
  DWORD WINAPI MyThread1(LPVOID param)
  {
    int i;
    HDC hdc;

    // 호출이 허가되기를 기다린다.
    if(WaitForSingleObject(hSema, 10000)==WAIT_TIMEOUT) {
       MessageBox((HWND)param, "Time Out Thread 1",
                  "Semaphore Error", MB_OK);
       return 0;
    }

    for(i=0; i<MAX; i++) {

      if(i==MAX/2) {
        /* 처리가 절반 끝나면 신호기를 해제한다.
          이에 의해 Mythread2 의 실행이 가능하게 된다. */
        ReleaseSemaphore(hSema, 1, NULL);

        // 다시 호출이 허가되기를 기다린다.
        if(WaitForSingleObject(hSema, 10000)==WAIT_TIMEOUT) {
           MessageBox((HWND)param, "Time Out Thread 1",
                      "Semaphore Error", MB_OK);
           return 0;
        }
      }
```

```
    wsprintf(str, "Thread 1: loop # %5d ", i);
    hdc = GetDC((HWND) param);
    TextOut(hdc, 1, 1, str, lstrlen(str));
    ReleaseDC((HWND) param, hdc);
  }

  ReleaseSemaphore(hSema, 1, NULL);

  return 0;
}

// 프로세스안에서 실행되는 다른 하나의 스레드
DWORD WINAPI MyThread2(LPVOID param)
{
  int i;
  HDC hdc;

  // 호출이 허가되기를 기다린다.
  if(WaitForSingleObject(hSema, 10000)==WAIT_TIMEOUT) {
    MessageBox((HWND)param, "Time Out Thread 2",
               "Semaphore Error", MB_OK);
    return 0;
  }

  for(i=0; i<MAX; i++) {
    wsprintf(str, "Thread 2: loop # %5d ", i);
    hdc = GetDC((HWND) param);
    TextOut(hdc, 1, 20, str, lstrlen(str));
    ReleaseDC((HWND) param, hdc);
  }

  ReleaseSemaphore(hSema, 1, NULL);

  return 0;
}
```

신호기프로그람의 상세

이 프로그람에서는 두개의 스레드를 순차화하기 위하여 신호기의 손잡이가 hSema에 보관된다.

실행을 개시하면 MyThread2( )보다 먼저 MyThread1( )이 능동상태로 된다. MyThread1( )이 작성되였을 때 그것이 곧 신호기를 얻고 실행을 개시하는것이다. MyThread2( )가 작성되였을 때는 신호기를 얻을수 없으며 대기상태로 된다.

계속 대기상태로 있다가 MyThread1( )의 for순환이 MAX/2까지 실행되면 신호기가 해제되고 그로 인하여 MyThread2( )가 신호기를 받아서 실행을 개시한다. 이때는 MyThread1( )이 대기상태로 된다. MyThread2( )의 실행이 끝나고 신호기가 해제되면 MyThread1( )의 실행이 재개된다.

여기에 몇가지 시험해 보아야 할것들이 있다. 우선 신호기는 동시에 신호기를 호출할수 있는 스레드를 한개만으로 제한하고 있으므로 이것을 2 값신호기를 사용하는 방법으로 바꾸어보는것이다. 다음 hSema에 설정하는 계수기의 값을 2나 3으로 늘이고 스레드의 여러개의 실체를 실행할수 있게 하여 결과를 확인해 보는것이다.

사건객체의 사용방법

이미 설명한것처럼 *사건객체*는 사건의 발생을 스레드나 프로세스에 통지하는데 사용된다. 사건객체를 작성하기 위하여 *CreateEvent( )*라는 API함수를 사용한다. 선언은 다음과 같다.

```
HANDLE CreateEvent(LPSECURITY_ATTRIBUTES lpSecAttr,
                   BOOL Manual,
                   BOOL Initial,
                   LPCSTR lpszName);
```

lpSecAttr에는 보안속성 혹은 NULL을 설정한다. NULL을 설정한 경우는 체계설정의 보안서술자가 사용된다. Manual은 사건이 발생했을 때의 사건객체의 동작을 결정한다.

Manual의 값이 령이 아닌 경우 사건객체는 *ResetEvent( )*함수를 호출했을 때만 재설정되게 된다. 이 값이 령인 경우 사건객체는 차단된 스레드에 호출이 허가되였을 때 자동적으로 재설정된다.

Initial에는 객체의 초기상태를 설정한다. Initial의 값이 령이 아닌 경우 사건객체가 설정상태(사건이 통지된 상태)로 된다. 령인 경우는 사건이 재설정된 상태(사건이 통지되지 않은 상태)로 된다.

lpszName에는 사건객체의 이름을 표시하는 문자렬을 설정한다. 사건객체는 여러개

의 프로세스에서 사용되는 대역객체이다. 두개의 프로세스가 같은 이름으로 사건객체를 열기한 경우에는 두 프로세스가 동일한 객체를 참조한다. 이에 의해 두 프로세스가 순차화된다. lpszName에 NULL을 설정한 경우 객체는 한 프로세스안에서만 사용되게 된다.

*CreateObject( )*는 호출이 성공하면 사건객체의 손잡이를 돌려 주고 실패하면 NULL을 돌려 준다.

사건객체가 작성되면 사건의 발생을 기다리는 스레드(또는 프로세스)는 첫 파라메터에 사건객체의 손잡이를 주어 WaitForSingleObject( )를 호출한다. 이에 의해 스레드 또는 프로세스의 실행은 사건이 발생할 때까지 정지된다.

사건의 발생을 통지하는데는 *SetEvent( )*함수를 사용한다. 선언은 다음과 같다.

```
BOOL SetEvent(HANDLE hEventObject);
```

hEventObject 에는 기존의 사건객체의 손잡이를 설정한다. 이 함수가 호출되면 사건의 발생을 대기하고 있는 첫 스레드 또는 프로세스의 WaitForSingleObject( )가 돌림값을 돌려 주므로 실행을 재개할수 있다.

사건객체의 기능을 시험해 보기 위해 앞에서 작성한 프로그람을 다음과 같이 개조해 보자. 우선 hEvent라는 이름의 대역변수를 선언한다. 다음 WM_CREATE의 case문에 다음의 행을 추가한다.

```
hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
```

마지막으로 MyThread1( )과 MyThread2( )를 다음과 같이 변경한다.

```
// 첫번째 스레드
DWORD WINAPI MyThread1(LPVOID param)
{
  int i;
  HDC hdc;

  // 호출이 허가되기를 기다린다.
  if(WaitForSingleObject(hSema, 10000)==WAIT_TIMEOUT) {
     MessageBox((HWND)param, "Time Out Thread 1",
                "Event Error", MB_OK);
     return 0;
  }

  for(i=0; i<MAX; i++) {
    wsprintf(str, “Thread 1: loop # %5d “, i);
    hdc = GetDC((HWND) param);
```

```
        TextOut(hdc, 1, 1, str, lstrlen(str));
        ReleaseDC((HWND) param, hdc);
    }

    return 0;
}

// 두번째 스레드
DWORD WINAPI MyThread2(LPVOID param)
{
    int i;
    HDC hdc;

    for(i=0; i<MAX; i++) {
        wsprintf(str, "Thread 2: loop # %5d ", i);
        hdc = GetDC((HWND) param);
        TextOut(hdc, 1, 20, str, lstrlen(str));
        ReleaseDC((HWND) param, hdc);
    }

    // 사건을 통지한다.
    SetEvent(hEvent);

    return 0;
}
```

프로그람을 실행하면 MyThread2( )의 실행이 끝났다는것이 통지될 때까지 MyThread1( )의 실행이 차단된다.

기다림시계의 사용방법

일반적인 시계는 모든 Windows 판본에서 제공되지만 *기다림시계*는 최근에 와서 추가되였다. 표준시계를 신호기와 함께 사용할수도 있지만 기다림시계를 사용하는 편이 더 편리하다.

기다림시계는 놀라운 능력을 가지고 있다. 왜냐하면 과제를 배경으로 실행하는 기능

을 간단히 실현할수 있기때문이다.

이식과 관련한 요점 : 기다림시계는 Windows NT 4.0 에서부터 추가된것이며 Windows NT 3.51, Windows 95 및 Windows 3.1 에서는 지원되지 못하였다. 그러나 Windows 98 에서는 지원였다.

기다림시계는 설정된 시간까지 배경으로 실행되는 시계이다. 스레드는 사건객체의 경우와 같이 WaitForSingleObject( )함수를 사용하여 시계의 통지를 대기할수 있다. 스레드는 시계에 설정된 시간이 경과할 때까지 차단된다.

기다림시계를 작성하자면 *CreateWaitableTimer( )*함수를 사용하여야 한다. 선언은 다음과 같다.

```
HANDLE CreateWaitableTimer(LPSECURITY_ATTRIBUTES
                           lpSecAttr, BOOL manual,
                           LPCSTR lpszName);
```

lpSecAttr 에는 기다림시계의 보안서술자의 지시자를 설정한다. 이 파라메터에 NULL 을 설정하면 체계설정의 보안서술자가 사용된다.

manual 에 령이 아닌 값을 설정한 경우 시간이 경과한후에 시계를 수동적으로 재설정하여야 한다. 이 값에 령을 설정한 경우는 시계가 자동적으로 재설정된다.

lpszName 에는 시계의 이름을 설정한다. 시계에 이름을 붙이지 않은 경우는 이 파라메터에 NULL 을 설정한다. 이름을 달아 준 시계는 여러 개의 프로세스에서 공유될수 있게 된다. 이름을 붙이지 않은 시계는 그것을 작성한 프로세스내에서만 사용된다. 이 함수는 호출이 성공하면 기다림시계의 손잡이를 돌려 주며 호출이 실패하면 NULL 을 돌려 준다.

시계가 작성된 직후 시계는 능동상태로 되여 있지 않다. 시계를 설정하려면 *SetWaitableTimer( )*을 호출하여야 한다. 선언은 다음과 같다.

```
BOOL SetWaitableTimer(HANDLE hWaitTimer,
                      const LARGE_INTEGER *TargetTime,
                      LONG period,
                      PTIMERAPCROUTINE lpTimerFunc,
                      LPVOID param,
                      BOOL Unsuspend);
```

hWaitTimer 에는 시계객체의 손잡이를 설정한다. TargetTime 에는 시계의 완료시간

을 설정한다. period 에는 시계가 통지를 진행하는 시간간격을 ㎳단위로 설정한다. period에 령을 설정하면 시계로부터의 통지는 한번만으로 된다.

lpTimerFunc 에는 시계가 통지를 진행할 때 호출되는 함수(시계함수)의 지시자를 설정한다. 이 함수는 추가선택항목이다. 시계함수가 필요 없으면 lpTimerFunc 에 NULL 을 설정한다.

param에는 시계함수에 전달하는 파라메터를 설정한다. Unsuspend의 값이 령 아닌 경우는 전력절약방식으로 동작하고 있는 콤퓨터가 재개된다. SetWaitableTimer( )는 호출이 성공하면 령이 아닌 값을 돌려 주며 호출이 실패하면 령을 돌려 준다.

lpTimerFunc에 지적하는 시계함수의 선언은 다음과 같이 하여야 한다.

```
VOID CALLBACK TimerFunc(LPVOID param,
                        DWORD dwLowTime,
                        DWORD dwHightTime);
```

param 에는 SetWaitableTimer( )로부터 전달된 값이 보관된다. dwLowTime 과 dwHightTime 에는 *FILETIME* 구조체와 호환성 있는 형식으로 완료시간이 보관된다.(시간의 형식에 대해서는 뒤에서 본다.) 일반적으로 기다림시계를 사용하는 대부분의 응용프로그람에서는 시계함수를 필요로 하지 않는다.

SetWaitableTimer( )를 호출할 때는 목적하는 시간을 *LARGE_INTEGER* 공용체에 설정한다. 이 공용체는 *FILETIME* 구조체에서 정의되여 있는것과 호환성이 있는 64bit 옹근수이며 시간을 표시한다. 량자의 정의를 아래에 보여 주었다.

```
typedef struct _FILETIME {
   DWORD dwLowDateTime;   // 아래자리 32bit
   DWORD dwHightDateTime;  // 웃자리 32bit
}  FILETIME;

typedef union _LARGE_INTEGER {
   struct {
      DWORD LowPart;
      LONG HighPart;
   }
   LONGLONG QuadPart;
}  LARGE_INTEGER;
```

FILETIME구조체에는 1601년 1월 1일부터 경과한 시간이 100㎱의 단위로 보관된다. Win32 는 FILETIME 구조체에 보관된 시간을 다른 시간형식으로 변환하는 함수를

여러개 제공하고 있다. 기다림시계에서 목적하는 시간을 설정하기 위한 가장 간단한 방법은 우선 SYSTEMTIME 구조체를 사용하여 시간을 설정한 다음 그것을 FILETIME 구조체로 변환하는것이다.(이렇게 하여 LARGE_INTEGER 공용체로 변환할수 있다.) 제 14 장에서 본것처럼 SYSTEMTIME 구조체는 다음과 같이 정의되여 있다.

```
typedef struct _SYSTEMTIME {
    WORD wYear;            // 년
    WORD wMonth;           // 월(1~12)
    WORD wDayOfWeek;       // 요일(0~6)
    WORD Day;              // 일(1~31)
    WORD wHour             // 시
    WORd wMinute           // 분
    WORD wSecond           // 초
    WORD wMilliseconds;    // 미리초
} SYSTEMTIME;
```

목적하는 시간을 설정하려면 SYSTEMTIME 구조체의 성원들을 초기화하고 다음 SystemTimeToFileTime( )을 호출하여 그 시간을 FILETIME 구조체로 변환한다.

*SystemTimeToFileTime( )*의 선언은 다음과 같다.

```
BOOL SystemTimeToFileTime(CONST SYSTEMTIME *lpSysTime,
                          FILETIME *lpFileTime);
```

이 함수는 호출이 성공하면 령이 아닌 값을 돌려 주며 호출이 실패하면 령을 돌려준다.

기다림시계를 사용하는 대부분의 경우에 SYSTEMTIME 구조체의 모든 성원을 수동적으로 설정할 필요는 없다. 보통 현재체계시각을 얻고 그것에 목적하는 시간을 더하면 된다. 실례로 1 시간의 시계를 작성하려면 현재체계시각을 얻고 그의 wHour 성원에 적절한 값을 더하면 될것이다. 현재체계시각을 얻자면 *GetSystemTime( )*을 사용해야 한다. 선언은 다음과 같다.

```
VOID GetSystemTime(SYSTEMTIME *lpSysTime);
```

현재의 체계시각은 lpSysTime 이 가리키는 구조체에 돌려 진다. 체계시각은 협정세계시(UTC)로 표시된다. 이것은 기본적으로 그리니치표준시(GMT)와 같다.

실례 15-5 의 프로그람은 기다림시계의 사용방법을 보여 주는 프로그람이다. [F2]건을 누르면 스레드가 작성된다. 이 스레드에서 기다림시계가 작성되고 목적하는 시간이

10 ㎳단위로 설정된다. 창문이 최소화되고 스레드가 시계의 시간경과를 기다린다. 시계의 설정시간이 경과하면 창문이 본래의 크기로 복귀되며 스레드의 실행이 재개된다.

_WIN32_WINNT 정의를 프로그람의 선두에 기입한 점에 주의를 돌려야 한다. 기다림시계는 최근에 추가된것이므로 적절한 머리부파일의 정보를 인용하자면 이 정의가 필요하게 되는 번역프로그람들도 있다.

실례 15-5. WaitableTimer 프로그람

```
// 기다림시계의 실례

/* 아래의 정의는 기다림시계의 API를
   사용하기 위해 필요한것이다. */
#define _WIN32_WINNT 0x0400

#include <windows.h>
#include "thread.h"

#define MAX 10000

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
DWORD WINAPI MyThread1(LPVOID param);

char szWinName[] = "MyWin"; // 창문클라스의 이름

char str[255]; // 표시할 문자렬을 보관한다.

DWORD Tid1; // 스레드ID

HANDLE hWaitTimer; // 신호기의 손잡이

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;
  HACCEL hAccel;
```

```
// 창문클라스를 정의한다.
wcl.cbSize = sizeof(WNDCLASSEX);

wcl.hInstance = hThisInst;      // 실체의 손잡이
wcl.lpszClassName = szWinName; // 창문클라스의 이름
wcl.lpfnWndProc = WindowFunc;  // 창문함수
wcl.style = 0;                 // 체계설정의 형식

wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

wcl.lpszMenuName = "WaitTimerMenu"; // 기본차림표

wcl.cbClsExtra = 0; // 보조기억기령역은 필요 없다.
wcl.cbWndExtra = 0;

// 창문의 배경색을 흰색으로 한다.
wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

// 창문클라스를 등록한다.
if(!RegisterClassEx(&wcl)) return 0;

/* 창문클라스가 등록되였으므로
   창문을 작성할수 있다. */
hwnd = CreateWindow(
  szWinName, // 창문클라스의 이름
  "Use a Waitable Timer", // 제목
  WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
  CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 너비는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 높이는 Windows가 결정하게 한다.
  NULL,          // 어미창문은 없다.
  NULL,          // 클라스차림표의 덧쓰기는 하지 않는다.
  hThisInst,     // 실체의 손잡이
  NULL           // 추가파라메터는 없다.
```

```
  );

  // 건반가속기를 적재한다.
  hAccel = LoadAccelerators(hThisInst, "WaitTimerMenu");

  // 창문을 표시한다.
  ShowWindow(hwnd, nWinMode);
  UpdateWindow(hwnd);

  // 통보문순환고리를 작성한다.
  while(GetMessage(&msg, NULL, 0, 0))
  {
    if(!TranslateAccelerator(hwnd, hAccel, &msg)) {
      TranslateMessage(&msg); // 건반통보를 변환한다.
      DispatchMessage(&msg); // Windows 2000 에 조종을 넘긴다.
    }
  }

  return msg.wParam;
}

/* 이 함수는 Windows 2000 으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받아 들인다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  int response;

  switch(message) {
    case WM_CREATE:
      // 시계를 작성한다.
      hWaitTimer = CreateWaitableTimer(NULL, 1, NULL);
      break;
    case WM_COMMAND:
      switch(LOWORD(wParam)) {
        case IDM_THREAD:
          CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)MyThread1,
```

```
                        (LPVOID) hwnd, 0, &Tid1);
              break;
            case IDM_EXIT:
              response = MessageBox(hwnd, "Quit the Program?",
                                    "Exit", MB_YESNO);
              if(response == IDYES) PostQuitMessage(0);
              break;
            case IDM_HELP:
              MessageBox(hwnd,
                         "F1: Help\nF2: Demonstrate Timer",
                         "Help", MB_OK);
              break;
          }
          break;
        case WM_DESTROY: // 프로그람을 끝낸다.
          PostQuitMessage(0);
          break;
        default:
          /* 이 switch 문에서 지정된것 이외의 통보문은
             Windows 2000 에 처리를 맡긴다. */
         return DefWindowProc(hwnd, message, wParam, lParam);
    }

    return 0;
}

// 기다림시계의 실례
DWORD WINAPI MyThread1(LPVOID param)
{
    int i;
    HDC hdc;
    SYSTEMTIME systime;
    FILETIME filetime;
    LARGE_INTEGER li;

    // 현재의 체계시각에 10s 를 더한다.
    GetSystemTime(&systime);
    SystemTimeToFileTime(&systime, &filetime);
```

```
    li.LowPart = filetime.dwLowDateTime;
    li.HighPart = filetime.dwHighDateTime;
    li.QuadPart += 100000000L;

      // 시계를 설정한다.
    SetWaitableTimer(hWaitTimer, &li,
                     0, NULL, NULL, 0);

    // 시계가 기동될 때까지 창문을 최소화해 둔다.
    ShowWindow((HWND) param, SW_MINIMIZE);

    // 시계의 기동을 대기한다.
    if(WaitForSingleObject(hWaitTimer, 100000)==WAIT_TIMEOUT) {
       MessageBox((HWND)param, "Time Out Thread 1",
                  "Timer Error", MB_OK);
       return 0;
    }

    // 경보음을 울리고 창문을 본래의 크기로 복귀한다.
    MessageBeep(MB_OK);
    ShowWindow((HWND) param, SW_RESTORE);

    hdc = GetDC((HWND) param);
    for(i=0; i<MAX; i++) {
      wsprintf(str, "Thread 1: loop # %5d ", i);
      TextOut(hdc, 1, 1, str, lstrlen(str));
    }
    ReleaseDC((HWND) param, hdc);

    return 0;
  }
```

이 프로그람은 앞의 실례프로그람과 같이 THREAD.H 라는 머리부파일을 사용한다. 그러나 이 프로그람이 사용하는 자원파일은 다음과 같다.

```
  #include <windows.h>
  #include "thread.h"

  WaitTimerMenu MENU
  {
    POPUP "&Options" {
```

```
      MENUITEM "&Waitable Timer\tF2", IDM_THREAD
      MENUITEM "E&xit\tCtrl+X", IDM_EXIT
    }
  MENUITEM "&Help", IDM_HELP
}

WaitTimerMenu ACCELERATORS
{
  VK_F1, IDM_HELP, VIRTKEY
  VK_F2, IDM_THREAD, VIRTKEY
  "^X", IDM_EXIT
}
```

기다림시계의 활용방법

기다림시계에는 여러가지 활용방법이 있다. 경종시계를 콤퓨터로 실현하는데 기다림시계를 사용한다. 그러자면 사용자가 목적하는 경종시각을 설정하기 위한 대화칸을 작성하고 설정된 값을 기다림시계를 초기화하는데 사용하면 편리할것이다.

기다림시계를 사용하여 자동적인 예비복사(Backup)나 파일전송 전용프로그람을 작성할수도 있다. 이러한 종류의 시계는 다른 동기화객체와 일반적인 시계를 결합하여 작성할수도 있지만 기다림시계를 리용하여 작성하는 방법이 보다 간단한것이다.

다른 프로그람을 기동하는 방법

Windows 2000 의 스레드에 기초한 다중과제처리기능은 프로그람작성기술의 양상에 커다란 충격을 주었다. 이밖에도 프로세스에 기초한 다중과제처리기능에는 여러가지 활용방법이 있다. 프로세스에 기초한 다중과제처리기능을 사용할 때는 같은 프로그람내에서 다른 스레드를 기동하는것이 아니라 한 프로그람이 다른 프로그람을 기동하는것으로 된다. Windows 2000 에서는 이것을 *CreateProcess( )*라는 API 함수를 사용하여 실현할 수 있다. 선언은 다음과 같다.

```
BOOL CreateProcess(LPCSTR lpszName, LPSTR lpszComLine,
                   LPSECURITY_ATTRIBUTES lpProcAttr,
                   LPSECURITY_ATTRIBUTES lpThreadAttr,
                   BOOL InheritAttr, DWORD How,
                   LPVOID lpEnv, LPCSTR lpszDir,
                   LPSTARTUPINFO lpStartInfo,
                   LPPROCESS_INFORMATION lpPInfo);
```

실행할 프로그람의 이름은 lpszName 이 가리키는 문자렬에 완전경로로 설정한다. 프로그람에 필요되는 지령행파라메터는 lpszComLine 이 가리키는 문자렬에 설정한다. 그러나 lpszName 에 NULL 을 설정하면 lpszComLine 에 설정된 문자렬의 첫 토막이 프로그람의 이름으로서 사용된다. 일반적으로는 lpszName 에 NULL 을 설정하고 프로그람의 이름과 필요한 파라메터를 lpszComLine 에 설정한다.(만일 Win16 의 프로세스를 기동하면 lpszName 에 NULL 을 설정하여야 한다.)

lpProcAttr 및 lpThreadAttr 에는 작성되는 프로세스의 보안속성을 설정한다. 이것들에 NULL 을 설정한 경우는 체계설정의 보안서술자가 사용된다. InheritAttr 에 령이 아닌 값을 설정한 경우는 프로세스의 작성에 사용된 손잡이가 새로운 프로세스에 계승된다. 이 파라메터에 령을 설정한 경우는 손잡이가 계승되지 않는다.

체계설정으로 새로운 프로세스는 표준적으로 실행된다. How 파라메터를 사용하면 프로세스의 작성방법에 영향을 주는 몇가지 속성을 설정할수 있다.(실례로 How 를 사용하여 프로세스에 특수한 우선권순위를 설정하거나 프로세스가 오유수정방식으로 동작한다는것을 지시할수도 있다.) How 에 령을 설정한 경우에 새로운 프로세스는 표준적인 프로세스로서 작성된다.

lpEnv 에는 새로운 프로세스의 환경파라메터를 보관한 완충기를 설정한다. lpEnv 에 NULL 을 설정한 경우 새로운 프로세스는 그것을 작성한 프로세스의 환경을 계승한다.

새로운 프로세스의 현재구동기(current drive)와 현재등록부(current directory)를 lpszDir 가 가리키는 문자렬에 설정한다. 이 파라메터에 NULL 을 설정하면 프로세스를 작성한 프로세스의 현재구동기와 현재등록부가 사용된다.

lpStartInfo 에는 새로운 프로세스의 기본창문의 표시방법을 가리키는 *STARTUPINFO* 구조체의 지시자를 설정한다. STARTUPINFO 구조체의 정의는 다음과 같다.

```
typedef struct _STARTUPINFO {
    DWORD cb;               // STARTUPINFO 의 크기
    LPSTR lpReserved;       // NULL 을 설정해야 한다.
    LPSTR lpDesktop;        // 탁상면의 이름
    LPSTR lpTitle;          // 조종탁의 이름(조종탁프로그람에서만)
    DWORD dwX;              // 창문의 왼쪽 웃모서리의 X 자리표
    DWORD dwY;              // 창문의 왼쪽 웃모서리의 Y 자리표
    DWORD dwXSize;          // 창문의 너비
    DWORD dwYSize;          // 창문의 높이
    DWORD dwXCountChars;    // 조종탁의 너비
    DWORD dwYCountChars;    // 조종탁의 높이
    DWORD dwFillAttribute;  // 본문과 배경의 색
    DWORD dwFlags;          // 어느 성원이 유효한가를 보여 주는 기발
    WORD wShowWindow;       // 창문의 표시방법(SW_SHOW 등)
    WORD cbReserved2;       // 0 을 설정해야 한다.
```

```
        LBYTE lpReserved2;        // NULL 을 설정해야 한다.
        HANDLE hStdInput;         // 이하는 표준손잡이
        HANDLE hStdOutput;
        HANDLE hStdError;
    }   STARTUPINFO;
```

dwX, dwY, dwXSize, dwYSize, dwXCountChars, dwYCountChars, dwFillAttribute, wShowWindow, hStdInput, hStdOutput 및 hStdError 의 각 성원은 dwFlags 성원에 설정된 값에 따라서 무시되거나 유효하게 된다. dwFlags 에 설정할 수 있는 값은 다음과 같다.

마 크 로	유효하게 되는 성원
STARTF_USESHOWWINDOW	dwShowWindow
STARTF_USESIZE	dwXSize, dwSize
STARTF_USEPOSITION	dwX, dwY
STARTF_USECOUNTCHARS	dwXCountChars, dwYCountChars
STARTF_USEFILLATTRIBUTE	dwFillAttribute
STARTF_USESTDHANDLES	hStdInput, hStdOutput, hStdError
STARTF_USEFORCEONFEEDBACK	유표를 표시한다.
STARTF_USEFORCEOFFFEEDBACK	유표를 표시하지 않는다.

dwFlags 에는 이 값들을 한개이상 조합하여 설정할수 있다.

참고 *: lpTitle, dwXCountChars, dwYCountChars 및 dwFillAttribute 의 각 성원은 조종탁응용프로그람에만 적용된다.*

일반적으로 STARTUPINFO 의 성원의 대다수는 설정할 필요가 없으므로 그것들을 무시하도록 dwFlag 를 설정한다. 그러나 구조체의 크기를 표시하는 cb 는 반드시 설정하고 기타 몇개의 성원에는 NULL 을 설정해야 한다는 점에 주의해야 한다.

CreateProcess( )의 마지막 파라메터인 lpPInfo 에는 *PROCESS_INFORMATION 구조체*의 지시자를 설정한다. 이 구조체의 정의는 다음과 같다.

```
    typedef struct _PROCESS_INFORMATION {
      HANDLE hProcess;       // 새로운 프로세스의 손잡이
      HANDLE hThread;        // 기본스레드의 손잡이
      DWORD dwProcessId;   // 새로운 프로세스의 ID
```

```
    DWORD dwThread;    // 새로운 스레드의 ID
} PROCESS_INFORMATION;
```

새로운 프로세스 및 그 프로세스의 기본스레드손잡이가 hProcess 및 hThread 에 돌려 진다. 새로운 프로세스 및 새로운 스레드의 ID 가 dwProcessId 및 dwThreadId 에 돌려 진다. 프로그람에서 이 정보를 사용하건 무시하건 관계 없다.

CreateProcess( )는 호출이 성공하면 령이 아닌 값을 돌려 주며 실패하면 령을 돌려 준다. 아래의 프로그람코드는 CreateProcess( )의 사용방법을 보여 주었다.

```
STARTUPINFO startinfo;
PROCESS_INFORMATION pinfo;
//...
startinfo.cb = sizeof(STARTUPINFO);
startinfo.lpReserved = NULL;
startinfo.lpDesktop = NULL;
startinfo.lpTitle = NULL;
startinfo.dwFlags = STARTF_USESHOWWINDOW;
startinfo.cbReserved2 = 0;
startinfo.lpReserved2 = NULL;
startinfo.wShowWindow = SW_SHOW;
CreateProcess(NULL, "test.exe",
              NULL, NULL, 0, 0,
              NULL, NULL, &startinfo, &pinfo);
```

여기에서는 어미프로세스의 현재등록부에 보관되여 있는 TEST.EXE 라는 이름의 프로그람을 기동하고 있다. 기동된 프로그람은 초기에 표시상태로 된다.

작성된 새로운 프로세스(새끼프로세스)는 그것을 작성한 어미프로세스와 거의 독립적인것으로 된다. 그러나 어미프로세스가 새끼프로세스를 완료시킬수 있다. 그렇게 하자면 *TerminateProcess( )*라는 API 함수를 사용해야 한다.

이식과 관련한 요점 : Windows 3.1 에서는 CreateProcess( )가 지원되지 않는다. Windows 3.1 에서 다른 프로세스를 기동하려면 WinExec( )함수를 사용한다. 그러나 WinExec( )은 낡은 형식의것이므로 Win32 프로그람에서는 사용하지 말아야 한다. 그러므로 낡은 프로그람을 이식할 때는 WinExec( )을 사용하는 부분을 CreateProcess( )로 치환하여야 한다.

제 16 장

두 종류의 도움말체계

Windows 2000 은 직결도움말(Online Help)을 폭넓게 지원한다. 일반적으로 모든 프로그람은 여러가지 기능에 대한 조작설명을 직결로 사용자에게 제공한다. 도움말체계(Help system)를 사용하여 직결문서를 제공하는 응용프로그람들도 많다. 직결도움말을 제공하지 않는 응용프로그람은 본격적인 Windows 2000 응용프로그람이라고 말할수 없다.

이 장에서는 직결도움말의 작성방법과 그것을 리용하기 위한 프로그람 작성기술에 대하여 설명한다.

도움말의 두 종류

Windows 2000 은 두가지 종류의 *도움말*을 지원하고 있다. 첫번째 도움말은 수년간 Win32 프로그람에서 사용되여 온 고전적인 수법을 리용하는것이다. 이 수법에서는 WinHelp( )라는 API 함수를 리용하여 일반적인 창문에 도움말정보를 표시한다. 이 수법은 일반적으로 WinHelp 도움말이라고 부른다.

두번째 도움말은 HTML 도움말이라고 하며 HtmlHelp( )라는 API 함수를 사용하는 수법이다. HTML 이란 HyperText Markup Language 의 줄임말이며 *HTML 도움말*에서는 열람기(Browser)형식으로 도움말정보를 표시한다.

이 장에서는 *WinHelp* 도움말과 *HTML 도움말*의 두가지 체계에 대해 설명한다. HTML 도움말방식은 새로운 방식으로서 Windows 2000 에서 받아 들인것이다. 고전적인 수법도 현존하는 많은 응용프로그람들에서 사용되고 있으며 많은 새로운 개발계획들에도 여전히 받아 들이고 있다. 그러므로 모든 프로그람작성자들은 우선 WinHelp 도움말에 대해 알아 둘 필요가 있다.

이 고전적인 도움말체계는 32bit 의 모든 판본의 Windows 에서 동작한다. 이에 비하여 HTML 도움말이 동작하자면 반드시 Internet Explorer(3.x 이후)의 부분품(Component)이 설치되여 있어야 한다. Windows 2000 은 이 조건을 완전히 만족시키고 있지만 Windows 95 등의 대다수 체계는 만족시키지 못하고 있다.

그러므로 이식성이 있는 도움말체계를 작성하자면 고전적인 수법을 사용해야 한다.

두 종류의 도움말체계는 내부적으로는 대체로 동일하며 도움말프로젝트의 작성방법에 차이가 있을뿐이다.

상황의존도움말과 참조도움말

Windows 가 지원하는 도움말은 두가지 부류로 구분할수 있다. 첫번째 부류는 직결문서인데 이것을 *참조도움말*이라고 부른다.

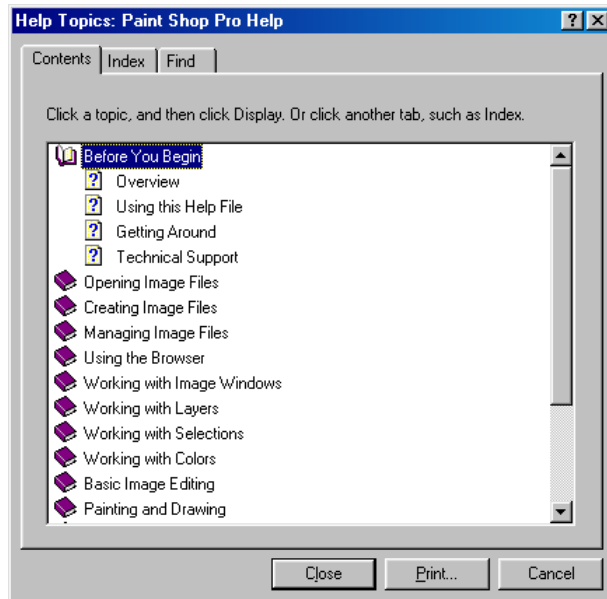
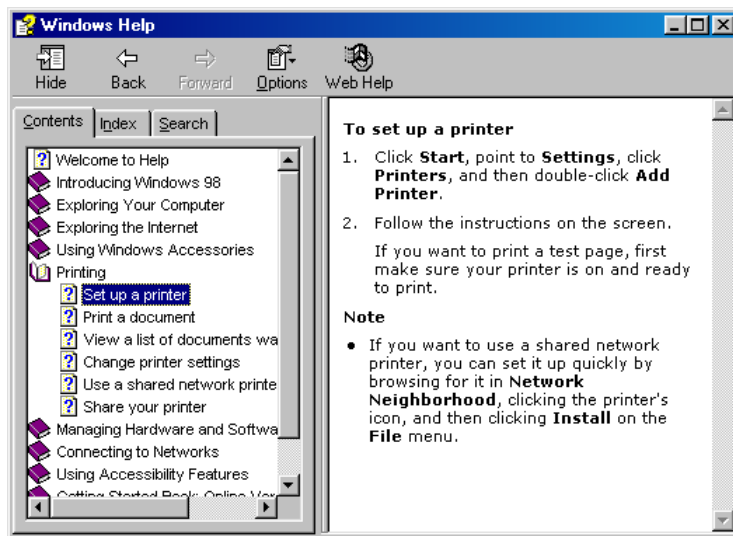
참조도움말은 그림 16-1 과 같은 도움말창문에 표시된것이다. 이 창문을 사용하여 여러가지 도움말표제(Topic)들을 표시하거나 다른 표제을 검색하거나 도움말파일의 차례를 표시할수 있다. 참조도움말은 응용프로그람이 지원하는 여러가지 기능에 대한 조작설명을 표시하거나 응용프로그람의 직결사용자지도서로서 리용된다.

ㄱ)

ㄴ)

그림 16-1. 도움말창문

ㄱ - 고전적인 도움말창문 , ㄴ - HTML 도움말의 창문

도움말의 두번째 부류는 *상황의존도움말*이다. *상황의존도움말*은 작은 창문안에 프로그

람의 부분적기능에 대한 짧은 설명을 표시하는데 쓰인다. 그림 16-2 에 그 실례를 보여 주었다. 본격적인 Windows 2000 응용프로그람으로 되자면 두 부류의 도움말기능을 갖추어야 한다. 겉보기에는 차이가 있지만 두가지 부류의 도움말은 대체로 동일한 방법으로 취급된다는것을 점차 알게 될것이다.

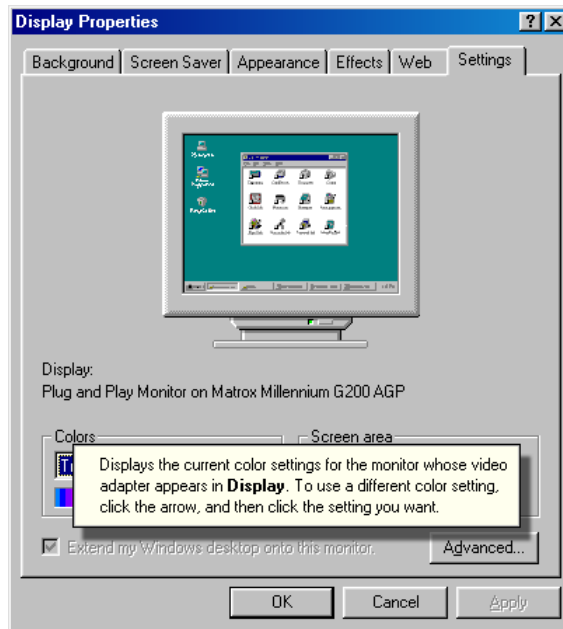



그림 16-2. 상황의존도움말의 실례

사용자가 도움말을 기동하는 네가지 방법

본격적인 Windows 응용프로그람을 작성하자면 사용자가 여러가지 방법으로 직결도움말을 기동할수 있어야 한다. 일반적으로 사용자는 아래의 네가지 방법으로 도움말을 기동한다.

- 객체우에서 마우스의 오른쪽 단추를 누른다.
- [?]단추를 누른 다음 객체를 찰칵한다.
- [F1]건을 누른다.
- [Help]차림표를 리용한다.

첫 두가지 방법에 의하면 상황의존도움말이 기동된다. 많은 경우 [F1]건도 상황의

존도움말을 기동하는데 사용되지만 참조도움말이 기동되는 경우도 있다. [Help]차림표를 사용하면 참조도움말이 기동된다.

임의의 시점에서 응용프로그람으로부터 도움말을 기동할수도 있다. 례하면 사용자가 무효한 조작을 진행한 경우에 응용프로그람이 도움말을 기동하는것과 같은 경우이다.

사용자가 도움말을 요구했을 때 응용프로그람은 사용하고 있는 도움말체계의 종류에 따라 *WinHelp( )* 또는 HtmlHlep( ) 의 어느 하나를 호출하여 응답한다.

WinHelp 에 의한 도움말체계의 리용방법

이 절에서는 *WinHelp( )*를 사용하는 고전적인 도움말체계를 설명한다. 이미 설명한 바와 같이 WinHelp( )는 고전적인(구식의) 도움말창문을 표시한다. 열람기형식의 도움말체계가 필요하지 않거나 혹은 리용할수 없는 환경에서 고전적인 도움말체계를 리용하여야 한다.

WinHelp 의 도움말파일

Windows 의 고전적인 도움말체계의 핵심으로 되는것은 *도움말파일*이다. 상황의존도움말과 참조도움말은 다 도움말파일을 리용한다. WinHelp 의 도움말파일은 본문파일이 아니다. 번역되여 있으며 .HLP 라는 확장자를 가지는 특수한 파일이다. 도움말파일을 작성하기 위해서는 먼저 RTF(Rich Text Format)파일을 작성하여야 한다. *RTF 파일*안에는 모든 도움말항목외에 양식정보, 색인정보, 교차참조정보 등이 보관된다. RTF 파일의 확장자는 일반적으로 .RTF 파일로 한다. RTF 파일은 도움말번역프로그람으로 번역한다.

WinHelp 의 도움말파일을 번역할 때는 Microsoft Help Workshop(프로그람파일의 이름은 HCW.EXE)를 리용한다. HCW.EXE 의 출력이 HLP 파일로 된다. 다시말하여 도움말번역프로그람의 위치에서 보면 RTF 파일이 원천파일이고 HLP 파일이 객체파일로 되게 된다.

RTF 파일을 서술하는 RTF 언어에는 수많은 지령들이 있다. 도움말번역프로그람은 이러한 지령들중의 일부에 대해서만 대응하고 있다. 또한 도움말체계에서는 일반적인 RTF 언어에 포함되여 있지 않는 특수한 지령도 몇가지 쓰인다. 모든 RTF 지령은 선두에 \표식이 붙어 있다. 실례로 \b 라는 RTF 지령은 굵은체(bold)를 의미한다.

이 장에서 모든 RTF 지령을 설명하는것은 불가능하다. 그래서 이 장에서는 도움말파일을 작성하는데서 가장 중요하게 쓰이는 지령들만을 설명한다. 만일 앞으로 WinHelp 의 도움말파일을 본격적으로 사용하려 한다면 RTF 지령을 전부 정통해야 한다.

도움말번역프로그람에는 RTF 파일뿐아니라 프로젝트파일도 필요하다. 이 파일은 HCW 도움말번역프로그람에 의해 작성된다. WinHelp 의 프로젝트파일의 확장자는 .HPJ

이다.

도움말파일의 작성

앞에서 설명한것처럼 WinHelp 의 도움말파일의 원천코드는 RTF 형식이여야 한다. 그러므로 도움말파일의 원천파일을 작성하는것은 결코 쉽지 않다.

도움말파일을 작성하는 방법에는 다음의 세가지의 선택가능성이 있다. 첫번째 방법은 도움말자동작성묶음을 구입하여 사용하는것이다. 두번째 방법은 RTF 파일을 작성하는 기능을 가진 본문편집기를 사용하는것이다. 세번째 방법은 일반적인 본문편집기를 사용하여 수동적으로 RTF 지령을 입력하는것이다.

만일 규모가 크고 복잡한 도움말파일을 작성한다면 도움말자동작성묶음을 사용하는것이 적절한 선택으로 된다. 그러나 작은 규모의 응용프로그람인 경우에는 나머지 두가지 방법으로도 충분하다. 여기서는 모든 사용자들이 쉽게 선택할수 있는 세번째 방법을 사용한다.

RTF 파일의 일반적인 서식

모든 RTF 파일에 공통된 기본서식이 있다. 우선 파일전체를 대괄호로 둘러 싸야 한다. 즉 *{ 로 시작하여 }*로 끝나야 한다. { 의 직후에는 \rtf 지령을 지정한다. 이 지령은 파일이 RTF 형식이라는것과 사용하는 RTF 의 판본을 표시한다.(여기에서는 판본 1 을 사용한다.)

다음 파일에서 사용하는 문자모임을 지정한다. 일반적으로 ANSI 문자모임을 사용하므로 여기에서는 \ansi 를 지정한다. 파일에서 쓰이는 서체도 지정하여야 한다. 이를 위해 \fonttbl지령을 사용한다. 그리하여 기본적인 RTF 파일의 서식은 아래와 같이 된다.

```
{\rtf1\ansi\fonttbl...
    Help File Contents
}
```

RTF 파일내부에서는 RTF 지령의 적용범위를 지적하기 위해 대괄호를 리용한다. 이 대괄호의 기능은 C/C++의 원천파일에서 대괄호로 블로크를 정의하는것과 같다.

주요한 RTF 지령들

간단한 도움말파일을 작성하는 경우에도 가장 중요하고 기본적인 몇개의 RTF 지령을 알고 있어야 한다. 이 장에서 사용하는 RTF 지령들을 표 16-1 에 주었다.

표 16-1. 도움말파일의 작성에서 사용되는 RTF 지령들

RTF 지령	기 능
\ansi	ANSI 문자모임을 지정한다.

\b	굵은체를 on 으로 한다.
\bO	굵은체를 off 로 한다.
\fn	n 으로 지정하는 서체를 사용한다.
\fsn	서체의 크기를 n 으로 한다.
\fonttbl	서체표를 정의한다.
\fontnote	열쇠단어 및 색인항목을 지정한다.
\i	경사체를 on 으로 한다.
\iO	경사체를 off 로 한다.
\page	도움말표제의 끝을 표시한다.
\par	단락의 끝을 표시한다.
\rtfn	사용하는 RTF 의 판본을 지적한다.
\tab	다음의 타브위치로 이동한다.
\uldb	다른 표제에로의 열점련결(Hot spot link)을 표시
\v	표제의 련결을 작성(\uldb 와 함께 사용)

\ansi

\ansi 지령은 ANSI 문자모임을 지정한다. RTF 파일은 \mac(Macintosh 문자모임)나 \pc(OEM 문자모임) 등 다른 문자모임들도 지원한다. 그러나 도움말파일에서 일반적으로 쓰이는것은 ANSI 문자모임이다.

\b

\b 지령은 굵은체를 on 으로 한다. \bO 지령은 굵은체를 off 로 한다. 그러나 \b 지령이 블로크안에서 사용되는 경우에는 블로크안에서 서술된 본문만이 굵은체로 되므로 \bO 를 사용할 필요는 없다. 아래에 실례를 보여 주었다.

```
{\b this is bold} this is not
```

이 실례에서는 대괄호안의 본문만이 굵은체로 된다.

\fn

\fn 지령은 서체를 선택한다. 서체는 번호로 선택한다. 선택되는 서체의 목록은 \fonttbl 지령을 사용하여 정의하여야 한다.

\fsn

\fsn 지령은 서체를 n 에서 지정한 크기로 한다. 크기는 0.5 포인트단위로 지정한다. 실례로 \fs24 는 서체의 크기를 12 포인트로 한다.

\fonttbl

서체를 사용하기에 앞서 사용할 서체를 \fonttbl 지령안에 렬거하여야 한다. 아래에 일반적인 서식을 보여 주었다.

```
{\fonttbl
    {\f1\family-name font-name;}
    {\f2\family-name font-name;}
    {\f3\family-name font-name;}

         ⋮

    {\fn\family-name font-name;}
    }
```

family-name은 서체계렬의 이름(froman 이나 fswiss 등)이며 font-name은 그 계렬에 부속된 특정한 서체의 이름(Times New Roman, Arial 및 Old English 등)이다. \f 지령으로 지정하는 서체의 번호는 서체를 선택하는데 쓰인다. 서체계렬이름의 실례를 아래에 보여 주었다.

서체계렬	서체의 이름
\froman	Times New Roman, Palatino
\fswiss	Arial
\fmodern	Courier New, Pica
\fscript	Cursive
\fdecor	Old English

다음의 실례는 0번의 서체를 \fswiss Arial로 하는것이다.

```
{\fonttbl {\f0\fswiss Arial;}}
```

\footnote

\footnote 지령은 도움말파일의 작성에 있어서 가장 중요한 RTF 지령의 하나이다. 왜냐하면 이 지령을 사용하여 표제이름, *상황ID* 및 *열람렬*(Browse sequence)이 설정되기때문이다. 이 장에서는 다음에 보여 주는 서식의 \footnote 지령을 사용한다.

```
${\footnote string}
```

K{\footnote string}
#{\footnote string}
+{\footnote sequence-name:sequence-order}
@{\footnote string}

$를 사용한 서식은 표제의 제목을 지정한다. 이 제목은 도움말체계의 경력창(History window)에도 표시된다. 제목은 공백을 포함할수 있다. 표제의 제목은 표제를 식별하기 위한것이다.

K를 사용한 서식은 string에 열쇠단어를 지정한다. 열쇠단어는 공백을 포함할수 있다. 열쇠단어는 색인항목으로서 표시된다. 열쇠단어의 선두문자가 K 인 경우는 그앞에 공백을 배치하여야 한다. K를 사용한 서식은 표제의 제목이 지정되여 있는 경우에만 사용된다.

#를 사용한 서식은 표제간의 련결이나 교차참조를 작성하는데 쓰이는 상황 ID 를 지정한다. 이 상황 ID 는 응용프로그람으로부터 도움말파일의 특정한 부분을 호출하는데도 쓰인다. #를 사용한 서식에서는 string에 공백을 포함할수 없다. 상황 ID로 되는 문자렬에는 ID의 값을 가리키는 마크로를 사용하는것이 일반적이다.

+를 사용한 서식은 열람순서렬을 지정한다. 열람순서렬이란 열람단추(도움말창의 단추띠에 표시되는 [<<] 단추 및 [>>]단추)가 눌리웠을 때 련결할 표제들을 결정한것이다.

+를 사용한 이 서식에서는 sequence-name 에 순서렬을 지정하고 sequence-order 에 순서렬안의 표제들의 위치를 지정한다. 열람순서렬은 sequence-order 의 값에 따라 영어자모순 또는 번호순으로 된다.

도움말파일은 한개이상의 열람순서렬을 가질수 있다. 여러개의 열람순서렬을 정의하는 경우에는 \footnote 지령의 sequence-name 과 sequence-order 를 지정하여야 한다. 도움말파일에 열람순서렬이 한개밖에 없는 경우는 sequence-name 을 지정할 필요가 없다.

표준적인 도움말창문에 열람단추를 표시하자면 열람순서렬을 정의한 도움말파일의 프로젝트파일의 [CONFIG]부분에 BrowseButtons( )마크로를 정의하여야 한다.

@를 사용한 서식은 설명문을 삽입할 때 사용한다.

\i

\i 지령은 경사체를 on 으로 설정한다. \iO 지령은 경사체를 off 로 설정한다. 그러나 \i 지령이 블로크내에서 사용되는 경우에는 블로크안에 서술된 본문만이 경사체로 되므로 \iO 지령을 사용할 필요가 없다.

\page

\page 지령은 대상항목의 끝을 표시한다.

\par

\ par 지령은 단락의 끝을 표시한다. 이 지령은 행을 바꾸는데 쓰인다. 실례로 한 행에 두개의 \ par 지령이 있으면 행바꾸기가 두번 진행된다.

\ rtfn
\ rtfn 지령은 사용하는 RTF 의 판본을 지적한다. 이 장에서는 판본 1 을 사용한다.

\ uldb
\ uldb 지령은 열점련결을 지정한다. 일반적인 서식은 다음과 같다.

\ uldb text
text 는 열점의 표준색과 서체로 표시된다. 이 지령은 흔히 \ v 지령과 함께 사용된다.

\ v
\ v 지령은 다른 다른 표제에로의 련결을 지정한다. 일반적인 서식은 다음과 같다.

\ v context-ID

context-ID 에는 #\ footnote 지령을 지정했을 때와 같이 상황 ID 를 지정한다. \ v 지령에서 지정된 열점을 사용자가 찰칵하면 련결이 진행된다.

WinHelp 에 대한 도움말파일의 실례

실례 16-1 의 도움말파일은 뒤에서 작성하는 WinHelp( )의 실례프로그람에서 리용된다. 이 파일에는 WinHelp 도움말파일의 기본적인 RTF 지령들을 모두 내포하고 있다. 이 파일의 이름은 HELPTEST.RTF 이다.

실례 16-1. Help 프로그람

```
{\rtf1\ansi
{\fonttbl{\f0\fswiss Arial;} {\f1\froman Times New Roman;}}
\fs40
\f1
@{\footnote This is a comment.  So is the following.}
@{\footnote This is a Sample Help File.}
${\footnote Contents}
Contents of Sample Help File
\f0
\fs20
```

```
\par
\par
\tab{\uldb Main Menu \v MenuMain}
\par
\tab{\uldb Main Window \v MainWindow}
\par
\tab{\uldb Main Window Push Button \v PushButtonMainWin}
\par
\tab{\uldb Push Button 1 \v PushButton1}
\par
\tab{\uldb Push Button 2 \v PushButton2}
\par
\tab{\uldb Push Button 3 \v PushButton3}
\par
\tab{\uldb List Box \v ListBox}
\par
\tab{\uldb Check Box \v CheckBox}
\par
\par
\f1
\fs30
Select a Topic.
\fs20
\f0
\page
#{\footnote MenuMain}
${\footnote Main Menu}
K{\footnote Main Menu}
{\fs24\b Main Menu}
\par
\par
The main menu allows you to display a sample
dialog box, activate the help system, display
information about this program, or terminate
the program.
\page
#{\footnote MenuMainPU}
This is the main menu for the program.
```

```
\page
#{\footnote PushButtonMainWin}
${\footnote Main Window Push Button}
K{\footnote Main Window Push Button}
{\fs24\b Main Window Push Button}
\par
\par
This is help for the main window push button.
\par
{\i This is in italics.}
\page
#{\footnote PushButtonMainWinPU}
This is the popup for the main window push button.
\page
#{\footnote PushButton1}
${\footnote Push Button 1}
K{\footnote Push Button 1}
+{\footnote Push:A}
{\fs24\b Push Button One}
\par
\par
This is help for the first push button.
\par
\par
See also {\uldb Push Button 2 \v PushButton2}
\page
#{\footnote PushButton1PU}
This is the popup for the first push button.
\page
#{\footnote PushButton2}
${\footnote Push Button 2}
K{\footnote Push Button 2}
+{\footnote Push:B}
{\fs24\b Push Button Two}
\par
\par
This is help for the second push button.
\par
```

```
\par
See Also {\uldb Push Button 3 \v PushButton3}
\page
#{\footnote PushButton2PU}
This is the popup for the second push button.
\page
#{\footnote PushButton3}
${\footnote Push Button 3}
K{\footnote Push Button 3}
+{\footnote Push:C}
{\fs24\b Push Button Three}
\par
\par
This is help for the third push button.
\par
\par
See Also {\uldb Push Button 1 \v PushButton1}
\page
#{\footnote PushButton3PU}
This is the popup for the third push button.
\page
#{\footnote MainWindow}
${\footnote Main Window}
K{\footnote Main Window}
{\fs24\b Main Window}
\par
\par
This is the main program window.
\page
#{\footnote MainWindowPU}
This is the main window popup help message.
\page
#{\footnote DlgPU}
${\footnote DlgPU}
This is a dialog box.
\page
#{\footnote ListBox}
${\footnote List Box}
```

```
K{\footnote List Box}
+{\footnote BOX:A}
{\fs24\b List Box}
\par
\par
This is help for the list box.
\page
#{\footnote ListBoxPU}
This is the popup for the list box.
\page
#{\footnote CheckBox}
${\footnote Check Box}
K{\footnote Check Box}
+{\footnote BOX:B}
{\fs24\b Check Box}
\par
\par
This is help for the check boxes.
\page
#{\footnote CheckBoxPU}
This is the popup for the check boxes.
\page
#{\footnote MenuDlgPU}
Activates the sample dialog box.
\page
#{\footnote MenuExitPU}
Termintes the program.
\page
#{\footnote MenuHelpPU}
Activates the Help System.
\page
#{\footnote MenuAboutPU}
Displays information about this program.
\page
}
```

이 파일에서는 ID 이름의 끝이 PU 로 되여 있는 #\footnote 지령이 상황의존도움말

의 입구점으로 되여 있다. 기타 #\footnote지령들은 표준적인 도움말창문에서 참조도움말을 표시하는데 쓰인다.

이 파일을 작성한 다음 HCW 번역프로그람을 리용하여 번역을 진행한다. 출력되는 파일은 HELPTEXT.HLP로 된다. HELPTEXT.HLP는 *WinHelp( )*의 실례프로그람에서 리용된다. 그러나 파일을 번역하기전에 HELPTEST.RTF의 프로젝트파일의 [MAP] 부분에 아래와 같은 MAP 명령문들을 정의하여야 한다. 이 값들은 상황의존도움말을 지원하는데 사용된다.

MAP 명령	값
PushButton1PU	700
PushButton2PU	701
PushButton3PU	702
ListBoxPU	703
CheckBoxPU	704
MainWindowPU	705
PushButtonMainWinPU	706
DigPU	707
MenuDlgPU	708
MenuExitPU	709
MenuHelpPU	710
MenuAboutPU	711
MenuMainPU	712

이렇게 많은 MAP명령들을 정의하기 위한 가장 간단한 방법은 도움말프로젝트에 매프파일을 포함시키는것이다. HCW의 경우는 매프파일안에서 값에 식별자를 대응시키기때문에 다음의 서식을 사용한다.

```
identifier1   value1
identifier2   value2
identifier3   value3
         ⋮
identifierN   valueN
```

례를 들면 실례도움말파일에서 사용되는 매프파일(이름이 HELPTEXT.HM인 본문파일)의 내용은 다음과 같다.

```
PushButton1PU           700
PushButton2PU           701
PushButton3PU           702
ListBoxPU               703
CheckBoxPU              704
MainWindowPU            705
PushButtonMainWinPU     706
DlgPU                   707
MenuDlgPU               708
MenuExitPU              709
MenuHelpPU              710
MenuAboutPU             711
MenuMainPU              712
```

표준도움말창에 열람단추를 표시하자면 프로젝트파일의 [CONFIG]부분에 BrowseButtons( )마크로를 지정하여야 한다. 도움말창문에 제목을 표시하자면 Help Workshop의 [Options]대화칸에서 설정한다. 제목을 설정하지 않은 경우에는 체계설정의 [Windows Help]라는 제목이 리용된다. 이 실례에서는 제목을 [Sample Help File]로 설정한다.

파일을 번역하면 HELPTEXT.HLP라는 도움말파일이 생성된다. 이 도움말파일은 후에 작성하는 WinHelp()의 실레프로그람의 EXE파일과 같은 등록부에 들어 있어야 한다.

다시 한보 전진

WinHelp의 마크로

BrowseButtons()마크로는 WinHelp의 도움말체계가 지원하는 많은 마크로들중의 하나에 불과하다. 기타 마크로들을 아래에 보여 주었다.

마크로	기능
Annotate( )	설명문칸을 표시한다.
Back( )	앞의 표제로 돌아 간다.
DisableButton(ButtonID)	ButtonID 에서 지정된 표준의 도움말단추를 무효로 한다.
EnableButton(ButtonID)	ButtonID 에서 지정된 표준의 도움말단추를 유효로 한다.
Exit( )	도움말체계를 완료한다.
CotoMark(mark)	SaveMark()로 설정된 바로 앞의 표식으로 이행한다.
History( )	경력목록을 표시한다.
Next( )	열람순서렬의 다음 표제로 전진한다. 유효한 열람순서렬안에서 실행되여야 한다.
Prev( )	열람순서렬의 앞표제로 돌아 간다. 유효한 열람순서렬내에서 실행되여야 한다.
Print( )	현재의 표제를 인쇄한다.
SaveMark( )	도움말파일의 현재위치를 mark에 지적한 표식으로 보관한다.

도움말마크로는 여러가지 방법으로 실행할수 있다. 실레로 BrowseButtons( ) 등의 마크로들은 도움말의 프로젝트파일안에서 지정되지만

도움말파일안에서 실행되는 마크로들도 있다. 도움말파일내에서 마크로를 실행하려면 다음과 같은 서식의 \footnote 지령을 리용한다.

!{\footnote macro( )}

macro 에 실행하려는 마크로의 이름을 지정한다. 실례로 아래의 지령은 열람순서렬의 앞 표제에로 돌아 가는 지령이다.

!{\footnote Prev( ) }

WinHelp 의 프로젝트를 작성할 때 도움말마크로를 사용하면 도움말정보를 사용자에게 표시하는 방법을 할수 있다. 앞으로 WinHelp 를 본격적으로 사용할수 있게 준비하자면 도움말의 마크로사용방법을 잘 알아 두어야 한다.

WinHelp( )를 사용한 도움말의 기동

RTF 파일을 작성하고 그것을 HLP 형식으로 번역하면 응용프로그람에서 WinHelp( )라는 API 함수를 불러 내여 도움말정보를 호출할수 있다. 아래에 이 API 함수의 선언을 준다.

```
BOOL WinHelp(HWND hwnd, LPCSTR filename, UINT command,
                DWORD extra);
```

hwnd에는 도움말을 기동하는 창문의 손잡이를 설정한다. filename에는 표시하려는 도움말파일의 이름을 구동기이름 및 경로이름과 함께 설정한다. command 에는 WinHelp( )함수에 보내려는 지령을 설정한다.

command에 설정할수 있는 지령들을 아래에 보여 주었다.

지　　령	동　　작
HELP_COMMAND	도움말마크로를 실행한다.
HELP_CONTENTS	(낡은 지령이므로 대신 HELP_FINDER 를 사용할것)
HELP_CONTENT	지정된 표제를 표시한다.
HELP_CONTEXTMENU	튀여나오기차림표를 표시한 다음 상황의존도움말을 표시한다.
HELP_CONTEXTPOPUP	상황의존도움말을 표시
HELP_FINDER	표준 도움말대상항목창을 표시
HELP_FORCEFILE	파일을 강제적으로 표시
HELP_HELPONHELP	도움말의 사용방법을 표시 WINHELP32.HLP 파일이 요구된다.
HELP_INDEX	(낡은 지령이므로 대신 HELP_FINDER 를 사용할것)
HELP_KEY	열쇠단어로 지정된 표제를 표시
HELP_MULTIKEY	다중열쇠단어로 지정된 표제를 표시
HELP_PARTIALKEY	열쇠단어와 부분적으로 일치하는 표제를 표시
HELP_QUIT	도움말창문을 닫는다.
HELP_SETCONTENTS	표제의 차례를 설정

HELP_SETPOPUP_POS	도움말체계에 의해 표시되는 튀여나오기창문의 위치를 설정
HELP_SETWINPOS	도움말창문의 크기를 설정하고 필요하다면 표시
HELP_TCARD	련습카드형식의 도움말을 사용하는 경우 이 지령을 다른 지령과 OR 로 결합하여 지정
HELP_WM_HELP	상황의존도움말을 표시

이 지령들가운데는 추가정보가 필요한것들도 있다. 추가정보는 extra 에 설정한다. 매 지령에서 extra 에 설정하는 값을 아래에 보여 주었다.

지　　령	extra 에 설정하는 값
HELP_COMMAND	마크로가 들어 있는 문자렬의 지시자
HELP_CONTENTS	미사용(령을 설정)
HELP_CONTENT	표제의 상황 ID
HELP_CONTEXTMENU	(본문을 참조)
HELP_CONTEXTPOPUP	표제의 상황 ID
HELP_FINDER	미사용(령을 설정)
HELP_FORCEFILE	미사용(령을 설정)
HELP_HELPONHELP	미사용(령을 설정)
HELP_INDEX	미사용(령을 설정)
HELP_KEY	열쇠단어가 들어 있는 문자렬의 지시자
HELP_MULTIKEY	MULTIKEYHELP 구조체의 지시자
HELP_PARTIALKEY	부분적인 열쇠단어가 들어 있는 문자렬의 지시자
HELP_QUIT	미사용(령을 설정)
HELP_SETCONTENTS	표제의 상황 ID
HELP_SETPOPUP_POS	POINTS 구조체의 지시자
HELP_SETWINPOS	HELPWININFO 구조체의 지시자
HELP_WM_HELP	(본문을 참조)

HELP_WM_HELP 와 *HELP_CONTEXTMENU* 에서는 extra 에 설정하는 값이 다른 지령보다 좀 복잡하다. 이 두 지령에서는 extra 에 DWORD 형의 배렬에 대한 지시자를 설정하는데 이 배렬에는 두개의 값을 설정한다. 첫번째 값은 조종체(누르기단추, 편집칸 등) 또는 차림표항목의 ID 를 설정한다.

두번째 값에는 그 조종체나 차림표항목과 관련된 도움말정보의 상황 ID 를 설정한다. 배렬의 끝에는 령을 두개 배치한다. 이 두개의 지령은 상황의존도움말을 지원하며 뒤에

서 설명하는 *WM_HELP* 통보문과 WM_CONTEXT 통보문을 처리할 때 사용된다.

WM_HELP 통보문과 WM_CONTEXT 통보문에 대한 응답

이 장을 시작할 때 설명한것처럼 도움말에는 크게 참조도움말과 상황의존도움말의 두가지 부류가 있다. 정확한 프로그람에서는 사용자가 차림표에서 [Help]를 선택하거나 또는 특정한 상황에서 [F1]건을 누르면 참조도움말이 표시된다.(표준의 도움말창이 표시된다.)

상황의존도움말은 조종체나 창문을 마우스의 오른쪽 단추로 찰칵하거나 *[?]단추*를 사용하거나 혹은 특정한 상황에서 [F1]건을 누르면 표시된다.([F1]건 사용방법에서의 차이점에 대해서는 다음 절에서 설명한다.)

프로그람은 각이한 조작의 도움말요청에 대해 각이한 방법으로 응답할수 있어야 한다. 그러므로 도움말요청을 구별하기 위한 어떤 수단이 필요하다. Windows 2000 은 이 수단을 정확히 제공하고 있다.

[F1]건이 눌리우든가 [?]단추가 리용되였을 때는 능동인 창문에 자동적으로 WM_HELP 통보문이 전송된다. 창문이나 조종체를 마우스의 오른쪽 단추로 눌렀을 때는 조종체를 포함하고 있는 창문에 *WM_CONTEXTMENU*통보문이 전송된다. 직결도움말을 정확하게 동작시키려면 이 두가지 통보문을 구별하여 처리하여야 한다. 여기에서는 두가지 통보문의 처리방법을 설명한다.

이식과 관련한 요점 : WM_HELP 통보문과 WM_CONTEXTMENU 통보문은 Windows 3.1 이나 Windows NT 3.51 에서는 지원되지 않는다. 그러므로 낡은 프로그람을 Windows 2000 에 이식할 때 필요에 따라 이 통보문들을 처리하는 기능을 추가하여야 한다.

WM_HELP 통보문

프로그람에 *WM_HELP* 통보문이 보내졌을 때 lParam 에는 도움말요청내용을 가리키는 *HELPINFO*구조체의 지시자가 보관되여 있다. *HELPINFO*구조체의 정의를 아래에 보여 주었다.

```
typedef struct tagHELPINFO
{
    UINT cbSize;
```

```
        int iContextType;
        int iCtrlId;
        HANDLE hItemHandle;
        DWORD dwContextId;
        POINT MousePos;
    }   HELPINFO;
```

cbSize 에는 HELPINFO 구조체의 크기가 보관된다.

iContextType 에는 도움말을 요청하고 있는 객체의 종류가 보관된다. 이 값은 차림표항목의 경우에는 HELPINFO_MENUITEM 으로 되며 창문 또는 조종체의 경우에는 HELPINFO_WINDOW 로 된다. iCtrlId 에는 조종체, 창문 또는 차림표항목의 ID 가 보관된다.

hItemHandle 에는 조종체, 창문 또는 차림표항목의 손잡이가 보관된다. dwContextId 에는 창문 또는 조종체의 상황 ID 가 보관된다. MousePos 에는 현재마우스의 위치가 보관된다.

많은 경우에 프로그람은 WM_HELP 통보문에 대한 응답으로서 상황의존도움말을 튀여나오기창문에 표시한다.

실례로 조종체의 상황도움말을 표시하자고 하면 창문손잡이를 가리키는 hItemHandle 의 값을 첫 파라메터로 하여 WinHelp( )를 호출한다.(이것은 목적하는 조종체의 손잡이이다.) 이 경우에는 WinHelp( )의 command 파라메터에 HELP_WM_HELP 를 설정하고 extra 파라메터에 ID 를 보관한 배렬의 지시자를 설정한다.(구체적인 방법에 대해서는 실레프로그람에서 보여 준다.)

이렇게 하여 WinHelp( )를 호출하면 우선 배렬로부터 hItemHandle 로 지정된 조종체와 일치하는 조종체 ID 가 검색된다. 그것에 대응하는 상황 ID 을 사용하여 상황의존도움말이 얻어 진다. 종국적으로 표준의 도움말창문이 아니라 튀여나오기창문에 도움말정보가 표시된다.(HtmlHelp( )를 사용하는 경우에도 이와 류사한 처리가 진행된다.)

WM_HELP 통보문에 대해 상황의존도움말을 표시하여 응답하는것은 일반적인 처리이지만 이 방법에 전혀 문제가 없는것은 아니다. 앞에서 설명한것처럼 [F1]건은 참조도움말 또는 상황의존도움말의 어느것을 호출하는데도 쓰인다. 여기서 [F1]건의 사용방법을 설명해 보자.

기본창문이 입력초점을 가지고 있을 때(다시말하여 어떤 새끼창문, 조종체 또는 차림표가 선택되여 있지 않을 때)에 [F1]건이 눌리우면 표준도움말창문이 기동되여 참조도움말이 표시된다. 그러나 조종체, 차림표 또는 새끼창문이 능동인 때 [F1]건이 눌리우면 상황의존도움말이 표시된다.

이것을 구별하는 리유는 사용자가 제일 웃준위의 창문에서 [F1]건을 눌렀을 때 프로그람전반에 대한 도움말을 요청한것이지 부분적인 도움말을 요청한것이 아니기때문이다. 이 요청에 대한 응답으로서 도움말체계전체를 기동한다. [F1]건이 눌리워 졌을 때

조종체(또는 새끼창문)가 능동으로 되여 있다면 사용자가 특정한 항목과 관련한 도움말을 요청한것이므로 상황의존도움말을 표시하게 된다.

[F1]건을 사용하여 참조도움말과 상황의존도움말가운데 어느것이나 기동할수 있으므로 프로그람에서 두가지 요청을 구별하려면 어떻게 하면 좋겠는가? [F1]건을 눌렀을 경우에 요청의 종류에 관계 없이 WM_HELP 통보문이 발송된다. 그러나 요청을 구별하는 방법은 아주 간단하다. hItemHandle 에 보관되여 있는 손잡이가 기본창문의것이라면 참조도움말을 표시하면 된다. 그렇지 않은 경우에는 앞에서 본 방법으로 상황의존도움말을 표시하면 좋다.

WM_CONTEXTMENU 통보문

사용자가 마우스의 오른쪽 단추를 찰칵하면 프로그람에 *WM_CONTEXTMENU* 통보문이 발송된다. 이때 wParam 에는 도움말의 대상으로 되는 조종체 또는 창문의 손잡이가 보관되여 있다. 사용자가 조종체를 오른쪽 찰칵한 경우는 창문의 손잡이로서 wParam 의 값을 지정하여 WinHelp( )를 호출한다.(WinHelp( )의 첫 파라메터는 창문의 손잡이이다.) 이때 command 파라메터에 *HELP_CONTEXTMENU* 를 설정하고 extra 파라메터에 ID 를 보관한 배렬의 지시자를 설정한다.(같은 방법이 HtmlHelp( )에서도 리용된다.)

[?]단추를 표시하는 방법

이미 설명한것처럼 상황의존도움말을 표시하기 위한 방법의 하나로서 *[?]단추*를 사용할수 있다. 창문에 [?]단추를 표시하자면 창문의 형식에 *WS_EX_CONTEXTHELP* 를 포함시켜야 한다. WS_EX_CONTEXTHELP 는 확장형식이므로 창문을 작성할 때 CreateWindow( )가 아니라 CreateWindowEx( )를 사용하여야 한다. 대화칸에 [?]단추를 표시하려는 경우에는 대화칸의 형식에 *DS_CONTEXTHELP* 를 포함시켜야 한다.

참고 *: 최신형식의 창문으로 하려면 기본적으로 대화칸에 [?]단추를 표시할것을 권고한다. 그러나 상황에 따라 기본창문에 [?]단추를 표시할수도 있다.*

WinHelp 의 실례프로그람

지금까지 설명한 고전적인 형식의 도움말체계를 작성하기 위한 여러가지 기술들을

실제로 사용해 보자. 실례 16-2 의 프로그람은 참조도움말과 상황의존도움말의 적용실례이다. 이 프로그람에서는 앞에서 작성한 도움말파일을 사용하며 WinHelp( )를 여러가지 방법으로 호출한다. 프로그람의 실행결과는 그림 16-3 과 같다.

실례 16-2. HelpTest 프로그람

```
// WinHelp 도움말체계의 실례

#include <windows.h>
#include "helptest.h"

#define NUMSTRINGS 6

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK DialogFunc(HWND, UINT, WPARAM, LPARAM);

char szWinName[] = "MyWin"; // 창문클라스의 이름

HINSTANCE hInst;

// 조종체의 ID 를 상황 ID 에 대응시킨다.
DWORD HelpArray[] = {
  IDD_PB1, IDH_PB1,
  IDD_PB2, IDH_PB2,
  IDD_PB3, IDH_PB3,
  ID_PB0, IDH_PB0,
  IDD_LB1, IDH_LB1,
  IDD_CB1, IDH_CB1, // 여기서는 두개의 검사칸에
  IDD_CB2, IDH_CB1, // 같은 상황 ID 를 대응시키고 있다.
  0, 0
};

char lbstring[6][40] = {
  "one", "two", "three",
  "four", "five", "six"
};

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
```

```
                    LPSTR lpszArgs, int nWinMode)
{
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;
  HACCEL hAccel;

  // 창문클라스를 정의한다.
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst;      // 실체의 손잡이
  wcl.lpszClassName = szWinName; // 창문클라스의 이름
  wcl.lpfnWndProc = WindowFunc;  // 창문함수
  wcl.style = 0;                 // 체계설정의 형식

  wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
  wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
  wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

  wcl.lpszMenuName = "HelpDemoMenu"; // 기본차림표

  wcl.cbClsExtra = 0; // 보조기억기령역은 필요 없다.
  wcl.cbWndExtra = 0;

  // 창문의 배경색을 흰색으로 한다.
  wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

  // 창문클라스를 등록한다.
  if(!RegisterClassEx(&wcl)) return 0;

  /* 창문클라스가 등록되였으므로
     창문을 작성할수 있다. */
  hwnd = CreateWindowEx(
    WS_EX_CONTEXTHELP, // [?]단추를 표시한다.
    szWinName, // 창문클라스의 이름
    "A WinHelp Demonstration", // 제목
    WS_SYSMENU | WS_SIZEBOX,
    CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
```

```
    CW_USEDEFAULT, // Y 자리표는 Windows 가 결정하게 한다.
    CW_USEDEFAULT, // 너비는 Windows 가 결정하게 한다.
    CW_USEDEFAULT, // 높이는 Windows 가 결정하게 한다.
    NULL,          // 어미창문은 없다.
    NULL,          // 클라스차림표의 덧쓰기는 하지 않는다.
    hThisInst,     // 실체의 손잡이
    NULL           // 추가파라메터는 없다.
  );

  hInst = hThisInst; // 현재의 실체손잡이를 보관한다.

  // 전반가속기를 적재한다.
  hAccel = LoadAccelerators(hThisInst, "HelpDemoMenu");

  // 창문을 표시한다.
  ShowWindow(hwnd, nWinMode);
  UpdateWindow(hwnd);

  // 통보문순환고리를 작성한다.
  while(GetMessage(&msg, NULL, 0, 0))
  {
    if(!TranslateAccelerator(hwnd, hAccel, &msg)) {
      TranslateMessage(&msg); // 건반통보를 변환한다.
      DispatchMessage(&msg); // Windows 2000 에 조종을 넘긴다.
    }
  }

  return msg.wParam;
}

/* 이 함수는 Windows 2000 으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받아 들인다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  int response;
```

```
switch(message) {
  case WM_CREATE:
    // 조종체(새끼창문)를 작성한다.
    CreateWindow(
      "BUTTON", // 조종체의 클라스이름
      "Main Window PB", // 제목
      BS_PUSHBUTTON | WS_CHILD | WS_VISIBLE, // 누름단추
      10, 60, 120, 30,
      hwnd, // 어미는 기본창문이다.
      (HMENU) ID_PB0, // 조종체의 ID
      hInst, // 프로그람의 실체손잡이
      NULL // 추가파라메터는 없다.
    );
    break;
  case WM_HELP: // 사용자가 [F1]건을 눌렀든가 [?]단추를 사용하였다.
    if(((LPHELPINFO) lParam)->iContextType ==
                                HELPINFO_MENUITEM) {
      // 차림표와 관련한 도움말요청
      switch(((LPHELPINFO) lParam)->iCtrlId) {
        case IDM_DIALOG:
            WinHelp(hwnd, "helptest.hlp", HELP_CONTEXTPOPUP,
                  (DWORD) IDH_MENUDLG);
          WinHelp(hwnd, "helptest.hlp>HlpWin2", HELP_CONTEXT,
                  (DWORD) IDH_MENUDLG);
          break;
        case IDM_HELP:
          WinHelp(hwnd, "helptest.hlp", HELP_CONTEXTPOPUP,
                  (DWORD) IDH_MENUHELP);
          break;
        case IDM_HELPTHIS:
          WinHelp(hwnd, "helptest.hlp", HELP_CONTEXTPOPUP,
                  (DWORD) IDH_MENUABOUT);
          break;
        case IDM_EXIT:
          WinHelp(hwnd, "helptest.hlp", HELP_CONTEXTPOPUP,
                  (DWORD) IDH_MENUEXIT);
          break;
        default:
```

```
                // 차림표띠가 선택되였으나 반전표시된 항목은 없다.
                WinHelp(hwnd, "helptest.hlp", HELP_CONTEXTPOPUP,
                        (DWORD) IDH_MENUMAIN);
            }
          }
          else
            if(((LPHELPINFO) lParam)->hItemHandle != hwnd) {
              // 조종체와 관련한 상황도움말
              WinHelp((HWND)((LPHELPINFO) lParam)->hItemHandle,
                      "helptest.hlp", HELP_WM_HELP,
                      (DWORD) HelpArray);
            }
          else {
            // 기본창문의 표준도움말
            WinHelp(hwnd, "helptest.hlp", HELP_KEY,
                    (DWORD) "Main Window");
          }
          break;
        case WM_CONTEXTMENU: // 사용자가 마우스의 오른쪽 단추를 찰칵하였다.
          if((HWND) wParam != hwnd)
            // 조종체와 관련한 상황도움말
            WinHelp((HWND) wParam, "helptest.hlp",
                    HELP_CONTEXTMENU, (DWORD) HelpArray);
          else
            // 기본창문과 관련한 상황도움말
            WinHelp(hwnd, "helptest.hlp",
                     HELP_CONTEXTPOPUP, IDH_MAIN);
          break;
        case WM_COMMAND:
          switch(LOWORD(wParam)) {
            case IDM_DIALOG:
              DialogBox(hInst, "HelpDemoDB",
                        hwnd, (DLGPROC) DialogFunc);
              break;
            case IDM_HELP: // 차림표로부터 도움말이 요청되였다.
               WinHelp(hwnd, "helptest.hlp", HELP_FINDER, 0);
               break;
            case IDM_HELPTHIS:
```

```
            MessageBox(hwnd, "Help System Sample Program V1.0",
                       "About", MB_OK);
            break;
          case IDM_EXIT:
            response = MessageBox(hwnd, "Quit the Program?",
                                  "Exit", MB_YESNO);
            if(response == IDYES) PostQuitMessage(0);
            break;
        }
        break;
      case WM_DESTROY: // 프로그람을 끝낸다.
        WinHelp(hwnd, "helptest.hlp", HELP_QUIT, 0);
        PostQuitMessage(0);
        break;
      default:
        /* 이 switch 문에서 지정된것 이외의 통보문은
           Windows 2000 에 처리를 맡긴다. */
        return DefWindowProc(hwnd, message, wParam, lParam);
    }
    return 0;
}

// 실례프로그람의 대화함수
BOOL CALLBACK DialogFunc(HWND hdwnd, UINT message,
                         WPARAM wParam, LPARAM lParam)
{
  int i;

  switch(message) {
    case WM_HELP: // 사용자가 [F1]건을 눌렀든가 [?]단추를 사용하였다.
      // 조종체와 관련한 상황도움말
      WinHelp((HWND)((LPHELPINFO) lParam)->hItemHandle,
              "helptest.hlp", HELP_WM_HELP,
              (DWORD) HelpArray);
      return 1;
    case WM_CONTEXTMENU: // 사용자가 마우스의 오른쪽 단추를 찰칵했다.
      if((HWND) wParam != hdwnd)
        // 조종체와 관련한 상황도움말
```

```
          WinHelp((HWND) wParam, "helptest.hlp",
                  HELP_CONTEXTMENU, (DWORD) HelpArray);
        else
          // 대화함수와 관련한 상황도움말
          WinHelp(hdwnd, "helptest.hlp",
                   HELP_CONTEXTPOPUP, IDH_DLG);
        return 1;
      case WM_COMMAND:
        switch(LOWORD(wParam)) {
          case IDCANCEL:
            WinHelp(hdwnd, "helptest.hlp", HELP_QUIT, 0);
            EndDialog(hdwnd, 0);
            return 1;
          case IDD_PB1:
            MessageBox(hdwnd, "Push Button 1",
                        "Button Press", MB_OK);
            return 1;
          case IDD_PB2:
            MessageBox(hdwnd, "Push Button 2",
                        "Button Press", MB_OK);
            return 1;
          case IDD_PB3:
            MessageBox(hdwnd, "Push Button 3",
                        "Button Press", MB_OK);
            return 1;
        }
        break;
      case WM_INITDIALOG: // 목록칸을 초기화한다.
        for(i=0; i<NUMSTRINGS; i++)
          SendDlgItemMessage(hdwnd, IDD_LB1,
              LB_ADDSTRING, 0, (LPARAM) lbstring[i]);
        return 1;
    }

    return 0;
  }
```

이 프로그람은 아래의 자원파일을 리용한다.

```
// 도움말체계의 실례
#include <windows.h>
#include "helptest.h"

HelpDemoMenu MENU
{
  POPUP "&Dialog"
  {
    MENUITEM "&Dialog\tF2", IDM_DIALOG
    MENUITEM "E&xit\tCtrl+X", IDM_EXIT
  }
  POPUP "&Help"
  {
    MENUITEM "&Help Topics", IDM_HELP
    MENUITEM "&About", IDM_HELPTHIS
  }
}

HelpDemoMenu ACCELERATORS
{
  VK_F2, IDM_DIALOG, VIRTKEY
  "^X", IDM_EXIT
}

HelpDemoDB DIALOGEX 10, 10, 140, 110
CAPTION "Help Demonstration Dialog"
STYLE WS_POPUP | WS_SYSMENU | WS_VISIBLE | DS_CONTEXTHELP
{
  DEFPUSHBUTTON "Button 1", IDD_PB1, 11, 10, 32, 14
  PUSHBUTTON "Button 2", IDD_PB2, 11, 34, 32, 14
  PUSHBUTTON "Button 3", IDD_PB3, 11, 58, 32, 14
  PUSHBUTTON "Cancel", IDCANCEL, 8, 82, 38, 16
  AUTOCHECKBOX "Check Box 1", IDD_CB1, 66, 50, 60, 30
  AUTOCHECKBOX "Check Box 2", IDD_CB2, 66, 70, 60, 30
  LISTBOX IDD_LB1, 66, 5, 63, 33, LBS_NOTIFY |
```

```
                WS_BORDER | WS_VSCROLL | WS_TABSTOP
}
```

머리부파일 HELPTEST.H의 내용을 아래에 보여 주었다.

```
#define IDM_DIALOG         100
#define IDM_EXIT           101
#define IDM_HELP           102
#define IDM_HELPTHIS       103

#define IDD_PB1            200
#define IDD_PB2            201
#define IDD_PB3            202
#define IDD_LB1            203
#define IDD_CB1            205
#define IDD_CB2            206

#define ID_PB0             300

#define IDH_PB1            700
#define IDH_PB2            701
#define IDH_PB3            702
#define IDH_LB1            703
#define IDH_CB1            704
#define IDH_MAIN           705
#define IDH_PB0            706
#define IDH_DLG            707
#define IDH_MENUDLG        708
#define IDH_MENUEXIT       709
#define IDH_MENUHELP       710
#define IDH_MENUABOUT      711
#define IDH_MENUMAIN       712
```

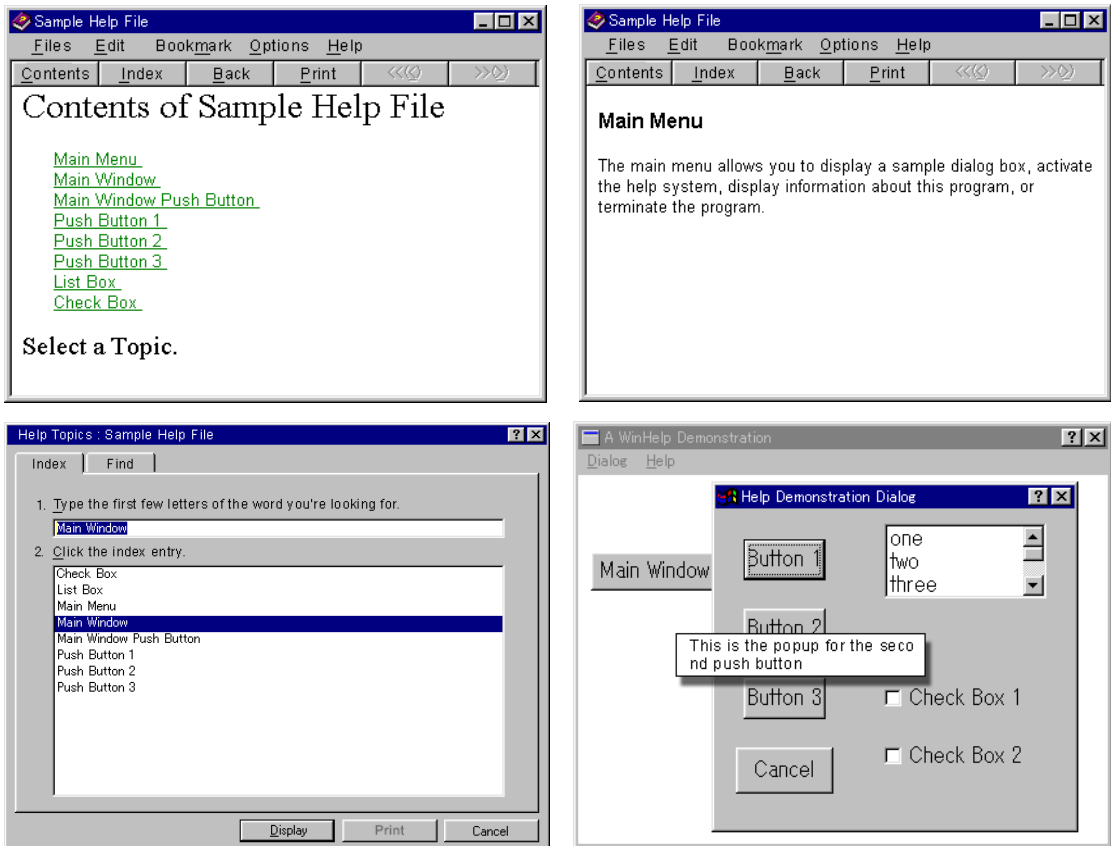



그림 16-3. WinHelp의 실례프로그람의 실행결과

WinHelp 실례프로그람의 상세

프로그람의 대부분의 내용은 쉽게 리해할수 있다. 그러나 주의해야 할 부분이 있다. 우선 HelpArray의 선언이다. 이 배렬은 조종체 ID와 상황 ID를 대응시키기 위한것이다.

HELPTEST.H 라는 머리부파일에서 IDH_로 시작되는 마크로들에는 도움말파일을 작성할 때 HELPTEST.PRJ 파일에서 정의한것과 같은 값들이 설정되여 있다. 두개의 검사칸에 같은 상황 ID가 대응되여 있는 점에도 주의를 돌려야 한다. 이렇게 한다고 해서 말썽거리가 생기지는 않는다. 그것은 두개이상의 조종체에 같은 상황의존도움말을 표시하는 경우에는 같은 도움말정보를 중복하여 작성할 필요가 없기때문이다.

이 프로그람에서 기본창문과 대화칸의 량쪽에 *[?]단추*를 표시하고 있는 점에 대해서도 주의를 돌려야 한다. [?]단추는 대화칸에서 사용하는것이 일반적이지만 기본창문에서도 사용한다. 여기에도 중요한 문제점이 있기때문이다. [?]단추를 눌러서 ?표식이 붙은 마우스는 그것이 표시되여 있는 창문에만 통보문을 보낸다. 실례프로그람을 사용하면 이것을 실제로 확인할수 있다.

우선 대화칸을 표시하고 [?]단추를 찰칵한다. 다음 마우스유표를 대화칸의 밖으로

이동한다. 마우스유표의 ?표식이 표시되지 않은것을 보게 될것이다. ?표식은 그것이 정의되여 있는 창문에만 표시된다.

또 한가지 주의를 돌려야 할 점이 있다. 기본창문의 내부에는 독립적인 누름단추가 있다. 이 단추는 기본창문의 새끼창문으로 되여 있다. 이 단추가 입력초점을 가지고 있을 때(선택되여 있을 때) [F1]건을 누르면 누름단추의 상황의존도움말이 튀여나오기 된다.

그러나 이 단추가 입력초점을 가지고 있지 않은 상태에서 [F1]건을 누르면 기본창문이 능동이므로 기본도움말창문이 표시된다. 이것은 Windows 의 표준설계지침에서 주장하는 도움말의 동작방식이다.

WinHelp 에서 2 차 창문을 리용하는 방법

WinHelp 프로젝트에는 도움말의 2 차 창문을 추가할수도 있다. 2 차 창문은 단추띠를 가지지만 차림표띠는 없다. 그러므로 표준의 참조도움말창문보다는 작고 상황의존도움말의 튀여나오기창문보다는 크게 된다. 2 차 창문은 그것을 닫을 때까지 능동으로 되여 있지만 사용자는 응용프로그람의 다른 부분을 계속 사용할수 있다.

2 차 창문을 작성하는 방법은 아주 간단하다. 우선 그것을 도움말프로젝트에 추가한다. 그러자면 HCW 번역기의 [Windows]추가선택항목에서 창문의 형태를 설정해 주기만 하면 된다. 2 차 창문의 종류에는 아래와 같은 세가지가 있다.

- Procedure
- Reference
- Error Message

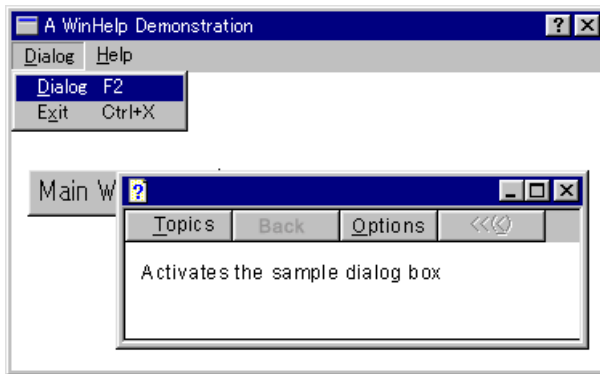
Procedure 창문은 [Help Topic], [Back] 및 [Options]라는 단추를 가지고 있는 작은 창문이다. 이 창문은 화면의 오른쪽 우에 표시된다. Reference 창문도 [Help Topic], [Back] 및 [Options]단추들을 가지고 있지만 약간 더 큰 창문으로 된다. 이 창문은 화면의 왼쪽 우에 표시된다. Error Message 창문은 단추가 없는 작은 창문이다.

2 차 창문을 사용하려면 WinHelp( )를 호출할 때 아래와 같은 서식으로 도움말파일의 이름과 창문이름을 설정한다.

```
filename>windowname
```

실례로 2 차 창문의 이름을 HlpWin2 로 한다면 아래의 코드로 앞의 실례프로그람의 기본차림표의 [Dialog]항목과 관련한 상황도움말을 2 차 창문에 표시할수 있다.

```
winHelp(hwnd, "helptest.hlp>HlpWin2", HELP_CONTEXT,
            (DWORD) IDH_MENUDLG);
```

HlpWin2 의 종류를 Reference 창문으로 하면 아래와 같은 창문이 표시된다.

도움말에 2차 창문을 추가하는 방법은 간단하지만 그렇게 하여 WinHelp 응용프로그람의 겉모양을 크게 개선할수 있다. 프로그람의 여러가지 장면에서 어떠한 형식의 창문을 사용하는것이 적절하겠는가 하는것은 자신들이 직접 시험해 보는것이 좋다.

HTML 도움말의 사용방법

WinHelp 에 기초한 도움말은 광범히 쓰이지만 그것을 강력한 *HTML 도움말*로 바꾸면 최신의 열람기지향대면부를 가진 도움말체계를 작성할수 있다.

HTML 도움말은 Windows 2000 에서 쓰고 있는 형식이다. HTML 도움말은 HTML 도움말표시기(HTML Help Viewer)에 의해 표시된다. 이것은 Internet Explorer 와 류사한 열람기형식의 창문이다. HTML 도움말의 례를 그림 16-4 에 보여 주었다.

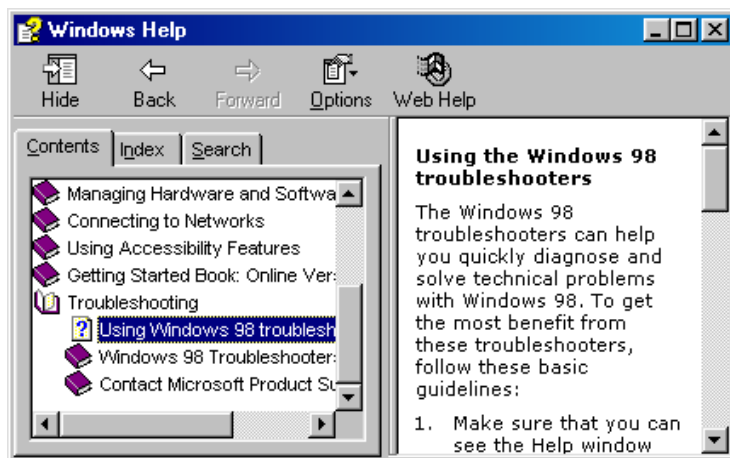



그림 16-4. HTML 도움말창문의 실례

HTML 도움말은 WinHelp( )가 아니라 HtmlHelp( )를 호출하여 표시한다. HTML 도움말을 리용하는 경우에도 프로그람은 지금까지 설명한 WM_HELP 통보문과 *WM_CONTEXTMENU* 통보문을 접수한다. 프로그람에서 변경이 필요한 개소는 이 통보문들에 응답하는 방법뿐이다.

HTML은 RTF가 아니다

HTML 도움말에서는 도움말정보를 보관하기 위하여 RTF 파일이 아니라 HTML 파일을 리용한다. 현재 대다수의 프로그람작성자들이 HTML에 대한 기초지식을 가지고 있으므로 여기서는 HTML에 대한 설명은 하지 않는다.

그러나 불과 몇개 안되는 HTML 지령만으로도 효과적인 도움말정보를 작성할수 있다는것을 알아 둘 필요가 있다. 즉 HTML에 대한 해박한 지식이 있으면 물론 편리하겠지만 꼭 필요한것은 아니다. 하이퍼련결(Hyper link)에 대한 지식만 있으면 HTML 도움말파일을 작성할수 있다.

HTML Help Workshop

본문편집기로 도움말파일을 간단히 작성할수 있는 고전적인 도움말체계와는 달리 HTML 파일을 작성하려면 Microsoft *HTML Help Workshop* (HHW.EXE)라는 쏘프트웨어가 있어야 한다. HTML Help Workshop는 HTML 도움말프로젝트를 구성하는 여러가지 파일이나 요소들을 관리할수 있다. 그림 16-5에 HTML Help Workshop를 보여주었다.

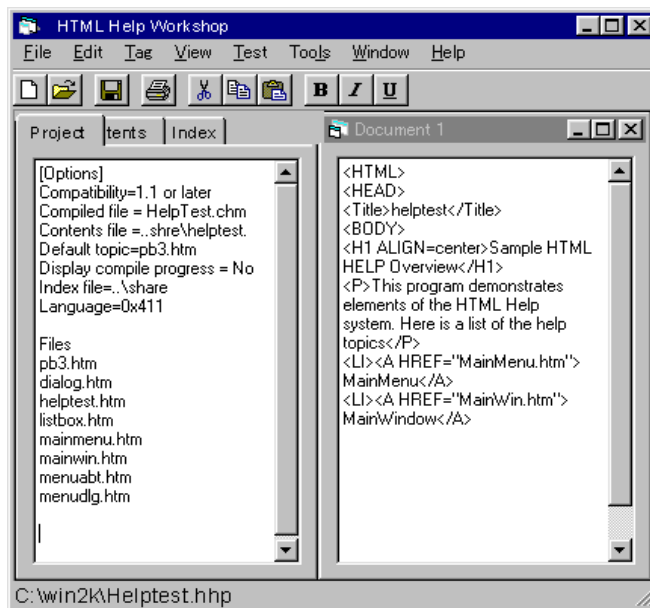



그림 16-5. HTML Help Workshop

상황 ID나 열쇠단어 등의 핵심적인 개념은 WinHelp나 HTML의 도움말프로젝트에

서 같지만 이 요소들을 프로젝트에 포함시키는 방법이 다르다. HTML Help Workshop 에는 조작법에 대한 상세한 직결도움말이 제공되여 있으므로 여기에서는 사용방법에 대해 설명하지 않는다.

HTML 도움말프로젝트는 여러개의 파일로 구성된다. 도움말정보를 보관하고 있는 HTML 파일, 상황의존도움말에 표시되는 문자렬을 보관하고 있는 본문파일, 색인파일 및 차례파일 등이다. 색인과 차례는 표제과 도움말정보를 보관하고 있는 HTML 파일의 련결을 지정하여 작성된다.

이러한 파일(HTML 파일도 포함)들은 모두 HTML Help Workshop 를 리용하여 작성할수 있다. 파일을 관리하는 프로젝트파일의 확장자는 .HHP 이다.

필요한 파일들을 작성하면 그것을 번역하여 *CHM 파일*을 작성한다. 이 파일에는 프로젝트에 정의되여 있는 모든 도움말정보가 보관되여 있다. *HtmlHelp( )*를 호출할 때 지정하는것이 이 파일이다. HTML Help Workshop 의 편리한 기능의 하나로 *CHM 파일*을 역번역하여 부분품(Component)단위로 분해하는 기능이 있다.

HTML Help Workshop 에는 WinHelp( )의 프로젝트를 HTML 도움말프로젝트로 변환하는 기능도 있다. 그러나 두 도움말체계의 차이를 고려하면 처음부터 HTML 도움말을 작성하는 편이 쉬울것이다.

HTML 도움말의 작성에 필요한 파일의 편집이나 관리는 HTML Help Workshop 를 사용하는것이 제일 적합하다. 여기에서는 매 파일들의 내용을 보여 주지 않는다.

HtmlHelp( )

HTML 도움말을 표시하려면 *HtmlHelp( )*를 호출하여야 한다. 우선 프로그람에 HTMLHELP.H 라는 머리부파일을 포함시켜야 한다. HTMLHELP.LIB 도 포함시켜야 한다. HtmlHelp( )의 선언을 아래에 준다.

```
HWND HtmlHelp(HWND hwnd, LPCSTR filename, UINT command,
                              DWORD extra);
```

hwnd 에는 도움말을 기동하는 창문의 손잡이를 지정한다. filename 에는 표시하려고 하는 도움말파일의 이름을 구동기이름 및 경로이름과 함께 지정한다. 이 파일은 HTML 도움말파일로서 번역되여 있으며 확장자가 CHM 이여야 한다. 이 파일은 여기에서 지정된 등록부안에 존재하여야 한다.

command 에는 HtmlHelp( )함수에 보내는 지령을 설정한다. command 에 설정할수 있는 지령들을 아래에 준다.

지　　령	동　　작
HH_DISPLAY_INDEX	HTML 도움말표시기에 [Keyword]표쪽을 표시한다. 제목을 filename 파라메터에

	지정한다.
HH_DISPLAY_SEARCH	HTML 도움말표시기에 [Find]표쪽을 표시한다. 표제를 filename 파라메터에 지정한다.
HH_DISPLAY_TOC	HTML 도움말표시기에 [Contents]표쪽을 표시한다. 제목을 filename 파라메터에 지정한다.
HH_DISPLAY_TOPIC	지정된 표제를 표시한다. 표제파일을 filename 파라메터에 지정한다.
HH_HELP_CONTEXT	ID로 지정된 표제를 표시한다.
HH_INITIALIZE	HTML 도움말을 초기화한다. hwnd 와 filename에 NULL을 설정한다. 프로그람의 기동시에 한번만 실행한다.
HH_TP_HELP_CONTEXTMENU	상황의존도움말을 표시한다. WM_CONTEXTMENU 통보문을 처리할 때 사용된다. 튀여나오기 되는 통보문을 보관한 파일을 filename 파라메터에 설정한다. 조종체의 손잡이를 hwnd 파라메터에 설정한다.
HH_TP_HELP_WM_HELP	상황의존도움말을 표시한다. 일반적으로 WM_HELP통보문을 처리할 때 발송된다. 튀여나오기되는 통보문을 보관하고 있는 파일을 filename 파라메터에 설정한다. 조종체의 손잡이를 hwnd 파라메터에 설정한다.
HH_UNINITIALIZE	HTML 도움말의 완료처리를 진행한다. hwnd와 filename에 NULL을 설정한다. 프로그람을 완료할 때 사용된다.

매 지령에서 extra에 설정하는 값을 아래에 준다.

지　　령	extra에 설정하는 값
HH_DISPLAY_INDEX	열쇠단어로 선택된 표제를 보관하고 있는 문자렬의 지시자
HH_DISPLAY_SEARCH	검색제목을 지정하는 HH_FTS_QUERY 구조체의 지시자
HH_DISPLAY_TOC	선택되는 표제를 보관하고 있는 문자렬의 지시

	자 또는 NULL
HH_DISPLAY_TOPIC	표시되는 표제를 보관하고 있는 문자렬의 지시자 또는 NULL
HH_HELP_CONTEXT	표제의 상황 ID
HH_INITIALIZE	Cookie 라고 하는 값을 받기 위한 DWORD 형의 지시자. 이 Cookie 는 HH_UNINITIALIZE 를 사용할 때 HtmlHelp( )에 전송된다.
HH_TP_HELP_CONTEXTMENU	(본문을 참조)
HH_TP_HELP_WM_HELP	(본문을 참조)
HH_UNINITIALIZE	HH_INITIALIZE 로 얻은 Cookie

HH_TP_HELP_WM_HELP 와 *HH_TP_HELP_CONTEXTMENU* 에서는 extra 에 DWORD 형의 배렬에 대한 지시자를 설정한다. 배렬에는 두개의 값을 설정한다. 첫번째 값에는 조종체(누름단추, 편집칸 등) 또는 차림표항목의 ID 를 설정한다. 두번째 값에는 그 조종체나 차림표항목과 관련한 도움말정보의 상황 ID 를 설정한다. 배렬의 끝에는 령을 두개 배치한다. 이 두개의 지령은 상황의존도움말을 지원한다. HtmlHelp( )는 호출이 성공하면 도움말창문의 손잡이를 돌려 주며 실패하면 NULL 을 돌려 준다.

HtmlHelp( )의 filename 파라메터의 상세

대다수의 지령에서는 HtmlHelp( )의 filename 파라메터에 CHM 파일의 이름을 설정한다. 그러나 일부 지령에서는 이 파라메터에 어떤 정보를 추가하기도 한다. 추가정보는 filename 이라는 하나의 문자렬안에서 두개의 점으로 구별되여 설정된다. 실례로 HH_DISPLAY_TOC 의 경우 filename 에 표시되는 CHM 파일과 함께 HTML 파일의 이름이 문자렬에 설정된다. 이 경우에 아래의 구문을 사용한다.

chm-filename::html-filename

아래에 보여 주는 *HtmlHelp( )*의 호출에서는 CHM 파일이 helptest.chm 이며 표제가 listbox.htm 으로 된다.

```
HtmlHelp(hwnd, “c:\\helptest.chm::/listbox.htm”, HH_DISPLAY_TOC, 0);
```

*CHM 파일*안에는 도움말프로젝트를 구성하는 파일들이 보관되여 있었던것과 꼭 같은 등록부구조가 들어 있다. 그러므로 실례로 listbox.htm 이 새끼등록부의 안에 보관되여 있었다면 그의 경로를 지정해 줄 필요가 있다.

HH_TP_CONTEXTMENU 또는 *HH_TP_HELP_WM_HELP* 지령을 사용하는 경우에

는 튀여나오기되는 본문을 보관하는 문자렬을 filename 파라메터에 설정한다. 례하면 HTML 도움말의 실례프로그람에서는 아래와 같은 코드를 사용하여 조종체의 도움말정보를 튀여나오기시키고 있다. 튀여나오기되는 본문은 htpopups.txt 라는 파일에 보관되여 있다.

```
// 조종체와 관련한 상황도움말
HtmlHelp( (HWND) wParam,
          "C:\\helptest.chm::htpopups.txt",
          HH_TP_HELP_CONTEXTMENU, (DWORD) HelpArray);
```

튀여나오기되는 본문은 아래와 같은 형식으로 설정한다.

```
.topic ID
popup text
```

ID 는 표제의 상황 ID 이다. 표시하려는 통보문은 popup text 에서 지정한다. 실례로 HTML 도움말의 실례프로그람에서 리용되는 파일의 맨 앞부분은 아래와 같다.

```
.topic IDH_PB0
This is the popup for the main window push button.
.topic IDH_PB1
This is the popup for the first push button.
.topic IDH_PB2
This is the popup for the second push button.
.topic IDH_PB3
This is the popup for the third push button.
```

여기에서는 구체적인 수값이 아니라 프로그람에서 정의한 마크로를 리용하고 있다는데 주의를 돌려야 한다. 그것은 HTML Help Workshop 에서는 도움말프로젝트에 표준의 C/C++머리부파일을 포함시킬수 있기때문이다.

filename 파라메터는 도움말의 2 차 창문을 지정하는데도 사용된다. 그러자면 우선 HTML Help Workshop 의 [Add/Modify window definition]추가선택항목을 사용하여 창문을 작성한다. 2 차 창문을 사용하기 위해서는 다음의 서식으로 filename 파라메터에 창문이름을 설정한다.

```
chm-filename>>windowname
```

HTML 도움말실레프로그람에서는 아래의 코드로 smallwin 이라는 이름의 2 차 창문에 [Dialog]차림표의 도움말정보를 표시하고 있다.

```
HtmlHelp( hwnd, “c:\\helptest.chm>>smallwin”,
                                   HH_HELP_CONTEXT, (DWORD) IDH_MENUDLG);
```

HtmlHelp 의 실례

아래의 프로그람은 WinHelp( )의 실레프로그람을 개조한것이다. 두 도움말체계는 같은 통보문을 처리하고 있으며 세부적처리는 거의 같으므로 변경이 필요한 부분은 많지 않다. 변경하는 내용은 다음과 같다.

- 머리부파일 HTMLHELP.H 를 포함시킨다.
- WinHelp( )의 호출을 HtmlHelp( )의 호출로 변경한다.

물론 HTML Help Workshop 를 사용하여 HTML 파일자체도 작성해야 한다. Htmlhelp( )의 실레프로그람의 프로그람코드를 실레 16-3 에 주었다. 이 프로그람을 실행하기에 앞서 이 책의 부속 CD-ROM 에 수록되여 있는 HELPTEST.CHM 을 C 구동기의 뿌리등록부에 복사하여 놓아야 한다. 왜냐하면 이 프로그람에서는 도움말파일의 경로를 고정하여 놓았기때문이다.(물론 이 경로이름을 변경하여도 상관없다.) HTMLHELP.LIB 도 련결하여야 한다. 프로그람의 실행결과를 그림 16-6 에 주었다.

실레 16-3. HtmlHelp 프로그람

```
HtmlHelp(hwnd, “C:\\helptest.chm>>smallwin”,
                                        HH_HELP_CONTEXT,
};

char lbstring[6][40] = {
  "one", "two", "three",
  "four", "five", "six"
};

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HWND hwnd;
  MSG msg;
```

```
WNDCLASSEX wcl;
HACCEL hAccel;

// 창문클라스를 정의한다.
wcl.cbSize = sizeof(WNDCLASSEX);

wcl.hInstance = hThisInst;      // 실체의 손잡이
wcl.lpszClassName = szWinName; // 창문클라스의 이름
wcl.lpfnWndProc = WindowFunc;  // 창문함수
wcl.style = 0;                 // 체계설정의 형식

wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

wcl.lpszMenuName = "HelpDemoMenu"; // 기본차림표

wcl.cbClsExtra = 0; // 보조기억기령역은 필요 없다.
wcl.cbWndExtra = 0;

// 창문의 배경색을 흰색으로 한다.
wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

// 창문클라스를 등록한다.
if(!RegisterClassEx(&wcl)) return 0;

/* 창문클라스가 등록되였으므로
   창문을 작성할수 있다. */
hwnd = CreateWindowEx(
  WS_EX_CONTEXTHELP, //  [?] 단추를 표시한다.
  szWinName, // 창문클라스의 이름
  "HTML Help Demonstration", // 제목
  WS_SYSMENU | WS_SIZEBOX,
  CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 너비는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 높이는 Windows가 결정하게 한다.
  NULL,          // 어미창문은 없다.
```

```
    NULL,          // 클라스차림표의 덧쓰기는 하지 않는다.
    hThisInst,     // 실체의 손잡이
    NULL           // 추가파라메터는 없다.
  );

  hInst = hThisInst; // 현재의 실체손잡이를 보관한다.
  // 건반가속기를 적재한다.
  hAccel = LoadAccelerators(hThisInst, "HelpDemoMenu");

  // 창문을 표시한다.
  ShowWindow(hwnd, nWinMode);
  UpdateWindow(hwnd);

  // 통보문순환고리를 작성한다.
  while(GetMessage(&msg, NULL, 0, 0))
  {
    if(!TranslateAccelerator(hwnd, hAccel, &msg)) {
      TranslateMessage(&msg); // 건반통보를 변환한다.
      DispatchMessage(&msg); // Windows 2000에 조종을 넘긴다.
    }
  }

  return msg.wParam;
}

/* 이 함수는 Windows 2000으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받아 들인다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  int response;
  static DWORD cookie = 0;

  switch(message) {
    case WM_CREATE:
      // HTML 도움말을 초기화한다.
      HtmlHelp(NULL, NULL, HH_INITIALIZE, (DWORD) &cookie);
```

```
        // 조종체(새끼창문)를 작성한다.
        CreateWindow(
          "BUTTON", // 조종체의 클라스이름
          "Main Window PB", // 제목
          BS_PUSHBUTTON | WS_CHILD | WS_VISIBLE, // 누름단추
          10, 60, 120, 30,
          hwnd, // 어미는 기본창문이다.
          (HMENU) ID_PB0, // 조종체의 ID
          hInst, // 프로그람의 실체의 손잡이
          NULL // 추가파라메터는 없다.
        );
        break;
      case WM_HELP: // 사용자가 [F1]건을 눌렀든가 [?]단추를 사용했다.
        if(((LPHELPINFO) lParam)->iContextType ==
                                    HELPINFO_MENUITEM) {
          // 차림표와 관련한 도움말요청
          switch(((LPHELPINFO) lParam)->iCtrlId) {
            case IDM_DIALOG:
              HtmlHelp(hwnd, "c:\\helptest.chm>smallwin",
                       HH_HELP_CONTEXT,
                       (DWORD) IDH_MENUDLG);
              break;
            case IDM_HELP:
              HtmlHelp(hwnd, "c:\\helptest.chm>smallwin",
                       HH_HELP_CONTEXT,
                       (DWORD) IDH_MENUHELP);
              break;
            case IDM_HELPTHIS:
              HtmlHelp(hwnd, "c:\\helptest.chm>smallwin",
                       HH_HELP_CONTEXT,
                       (DWORD) IDH_MENUABOUT);
              break;
            case IDM_EXIT:
              HtmlHelp(hwnd, "c:\\helptest.chm>smallwin",
                       HH_HELP_CONTEXT,
                       (DWORD) IDH_MENUEXIT);
              break;
```

```
          default:
            // 차림표띠가 선택되였으나 반전표시된 항목이 없다.
            HtmlHelp(hwnd, "c:\\helptest.chm>smallwin",
                     HH_HELP_CONTEXT,
                    (DWORD) IDH_MENUMAIN);
        }
      }
      else
        if(((LPHELPINFO) lParam)->hItemHandle != hwnd) {
          // 조종체와 관련한 상황도움말
          HtmlHelp((HWND)((LPHELPINFO) lParam)->hItemHandle,
                   "c:\\helptest.chm::htpopups.txt",
                   HH_TP_HELP_WM_HELP, (DWORD) HelpArray);
        }
        else {
          // 기본창문의 표준도움말
          HtmlHelp(hwnd, "c:\\helptest.chm>smallwin",
                   HH_HELP_CONTEXT, IDH_MAIN);
        }
      break;
    case WM_CONTEXTMENU: // 사용자가 마우스의 오른쪽 단추를 찰칵했다.
      if((HWND) wParam != hwnd)
        // 조종체와 관련한 상황도움말
        HtmlHelp((HWND) wParam,
                 "c:\\helptest.chm::htpopups.txt",
                 HH_TP_HELP_CONTEXTMENU, (DWORD) HelpArray);
      else
        // 기본창문과 관련한 상황도움말
        HtmlHelp(hwnd, "c:\\helptest.chm>smallwin",
                 HH_HELP_CONTEXT, IDH_MAIN);
      break;
    case WM_COMMAND:
      switch(LOWORD(wParam)) {
        case IDM_DIALOG:
          DialogBox(hInst, "HelpDemoDB",
                    hwnd, (DLGPROC) DialogFunc);
          break;
        case IDM_HELP: // 차림표로부터 도움말이 선택되였다.
```

```
            HtmlHelp(hwnd,
                     "c:\\helptest.chm::/helptest.htm",
                     HH_DISPLAY_TOC, 0);
            break;
        case IDM_HELPTHIS:
            MessageBox(hwnd, "HTML Help Sample Program V1.0",
                       "About", MB_OK);
            break;
        case IDM_EXIT:
           response = MessageBox(hwnd, "Quit the Program?",
                                 "Exit", MB_YESNO);
           if(response == IDYES) PostQuitMessage(0);
           break;
      }
      break;
    case WM_DESTROY: // 프로그람을 끝낸다.
      HtmlHelp(NULL, NULL, HH_UNINITIALIZE, cookie); // HTML 도움말을 닫는다.
      PostQuitMessage(0);
      break;
    default:
      /* 이 switch 문에서 지정된것 이외의 통보문은
         Windows 2000 에 처리를 맡긴다. */
      return DefWindowProc(hwnd, message, wParam, lParam);
  }
  return 0;
}

// 실례프로그람의 대화함수
BOOL CALLBACK DialogFunc(HWND hdwnd, UINT message,
                         WPARAM wParam, LPARAM lParam)
{
  int i;

  switch(message) {
    case WM_HELP: // 사용자가 [F1]건을 눌렀든가 [?]단추를 사용했다.
      // 조종체와 관련한 상황도움말
      HtmlHelp((HWND)((LPHELPINFO) lParam)->hItemHandle,
               "c:\\helptest.chm::htpopups.txt",
```

```
          HH_TP_HELP_WM_HELP,
          (DWORD) HelpArray);
  return 1;
case WM_CONTEXTMENU: // 사용자가 마우스의 오른쪽 단추를 찰칵했다.
  if((HWND) wParam != hdwnd)
    // 조종체와 관련한 상황도움말
    HtmlHelp((HWND) wParam,
             "c:\\helptest.chm::htpopups.txt",
             HH_TP_HELP_CONTEXTMENU, (DWORD)HelpArray);
  else
    // 대화함수와 관련한 상황도움말
    HtmlHelp(hdwnd, "c:\\helptest.chm>smallwin",
             HH_HELP_CONTEXT,
             (DWORD) IDH_DLG);
  return 1;
case WM_COMMAND:
  switch(LOWORD(wParam)) {
    case IDCANCEL:
      EndDialog(hdwnd, 0);
      return 1;
    case IDD_PB1:
      MessageBox(hdwnd, "Push Button 1",
                 "Button Press", MB_OK);
      return 1;
    case IDD_PB2:
      MessageBox(hdwnd, "Push Button 2",
                 "Button Press", MB_OK);
      return 1;
    case IDD_PB3:
      MessageBox(hdwnd, "Push Button 3",
                 "Button Press", MB_OK);
      return 1;
  }
  break;
case WM_INITDIALOG: // 목록칸을 초기화한다.
  for(i=0; i<NUMSTRINGS; i++)
    SendDlgItemMessage(hdwnd, IDD_LB1,
        LB_ADDSTRING, 0, (LPARAM) lbstring[i]);
```

```
        return 1;
    }

    return 0;
}
```

그림 16-6. HTML 도움말실례프로그람의 실행결과

자체로 해보기

도움말의 기능가운데서 프로그람작성자들이 절실히 요구하는 기능의 하나가 *련습카드*이다. 련습카드는 사용자에게 조작과 관련한 설명을 제공하기 위하여 사용된다. 례하면 문서편집기에서 단락을 설정하거나 인쇄방법 등을 선택하는 방법을 한걸음씩 설명하는데 련습카드를 사용할수 있다.

다음 WinHelp도움말의 프로젝트에서 *HelP_PARTIALKEY*라는 열쇠단어검색추가선

택항목을 사용해 볼수 있다. 편집칸을 표시하고 사용자에게 검색할 도움말정보의 열쇠단어를 입력시키고 그것과 부분적으로 일치하는 모든 표제들을 검색할수 있다.

HTML 도움말프로젝트에 검색기능도 추가해 볼수 있다.

마지막으로 한가지 보충해 둘것이 있다. 그것은 직결된 상황의존도움말이 모든 Windows 2000 응용프로그람에 있어서 중요한 구성요소이므로 후에 추가할것이 아니라 프로그람의 작성을 개시하는 시점에서 도움말지원기능을 고려해야 한다는것이다.

제 17 장

인쇄기의 사용법

Windows 2000 은 비트매프, 특수한 문자서체와 다양한 도형들을 광범히 리용하는 도형적인 조작체계이다. 그러므로 처음 Windows 2000 프로그람을 작성해 보는 사람들은 인쇄기능을 실현하려면 복잡하고 어려운 처리를 해야 하지 않겠는가고 생각할수 있다. 그러나 실제는 그렇지 않다.

Windows 2000 에 있어서 인쇄는 DOS 에 비하면 확실히 복잡한것이지만 Windows 2000 에는 고급한 인쇄를 실현하는 기능이 미리 갖추어 져 있다. Windows 2000 은 *인쇄기*를 조종하는 대부분의 기능을 제공하여 준다. 알아 두어야 할것은 많을수도 있지만 인쇄를 진행하는것 그자체는 결코 어렵지 않다. 기본적인 기술만 정통하면 모든 응용프로그람에 인쇄기능을 쉽게 추가할수 있다.

이 장에서는 다음의 기능을 실현하는 방법에 대해 설명한다.

- 본문인쇄
- 도형인쇄
- 도형의 척도설정
- 인쇄중지함수의 작성과 사용법

이것들은 Windows 2000 의 모든 인쇄기능의 기초로 된다. 설명을 하기에 앞서 한가지 알아 두어야 할것이 있다. 그것은 어떤 자료를 인쇄기에 출력하는 경우에 그것이 인쇄기 자체가 아니라 실제로는 인쇄완충기(Printer Spooler)에 전송된다는것이다. 이것은 프로그람작성방법에 직접 영향을 미치지는 않지만 알아 두어야 한다.

인쇄기장치상황의 얻기

화면장치상황(DC)이 화면에로의 출력을 관리하는것과 마찬가지로 인쇄기에로의 출력은 *인쇄기장치상황*에 의해 관리된다. 그러므로 인쇄기에 출력하기에 앞서 인쇄기의 장치상황을 얻어야 한다.

인쇄기장치상황을 얻는데는 여러가지 방법이 있다. 여기에서는 *CreateDC( )* 및 PrintDlgEx( )를 사용하는 방법을 설명한다. CreateDC( )는 사용자의 설정을 받지 않고 장치상황을 얻는다. PrintDlgEx( )는 공통대화칸을 표시하고 사용자에게 인쇄기장치상황의 설정을 진행하게 한다. 매 함수들의 사용방법을 설명해 보자.

CreateDC( )

인쇄기장치상황을 얻기 위한 첫번째 방법은 *CreateDC( )*를 사용하는것이다. 아래에 선언을 보여 주었다.

```
HDC CreateDC(LPCSTR lpszWhat, LPCSTR DevName,
                    LPCSTR NotUsed,
                    CONST DEVMODE *DevMode);
```

lpszWhat 에 "DISPLAY"(화면구동기를 얻는 경우) 또는 "WINSPOOL"(인쇄기구동프로그람을 얻는 경우)라는 문자렬지시자를 설정한다. 인쇄를 진행하는 경우 lpszWhat 에 "WINSPOOL"을 설정한다. DevName 에 인쇄기이름을 설정한다. NotUsed 에는 NULL 을 설정한다.

*CreateDC( )*는 호출이 성공하면 장치상황의 손잡이를 돌려 주며 실패하면 NULL 을 돌려 준다. 응용프로그람에서 인쇄처리를 완료하면 *DeleteDC( )*를 호출하여 인쇄기장치상황을 삭제해야 한다.

응용프로그람에서 현재 선택되여 있는 인쇄기의 이름을 얻어야 하는 경우도 있다. 그러자면 *EnumPrinter( )*를 사용하여 사용가능한 인쇄기의 이름을 렬거시킨다.

CreateDC( )는 사용자의 설정을 받지 않고 인쇄를 진행할 때 자주 사용되는 함수이다. 사용자에게 인쇄설정을 시키지 않는것은 일반적인것이 못될지도 모르지만 드문히 그런 정황이 조성된다. 례하면 사용자가 없을 때 체계리력기록을 인쇄하는 경우 등이다. 그러나 대부분의 경우(이 장의 실례에서도 같다.) 인쇄기장치상황을 얻는데는 *PrintDlgEx( )*가 사용된다.

PrintDlgEx( )

인쇄기장치상황을 얻기 위한 두번째방법은 *PrintDlgEx( )*를 사용하는것이다.

PrintDlgEx( )는 인쇄공통대화칸을 표시하고 사용자가 여러가지 추가선택항목들을 설정하게 한다. PrintDlgEx()는 Windows 2000에 새로 추가된것으로서 이전의 PrintDlg()를 치환한것이다.

파라메터를 설정하면 인쇄대화칸의 형태를 변경시킬수도 있지만 기본적으로는 그림 17-1에 표시한 형태로 된다.

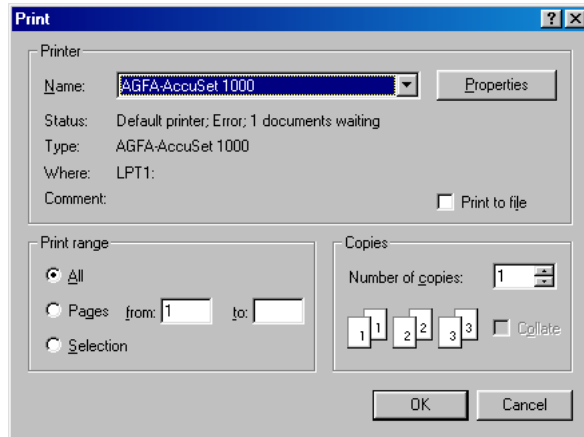



그림 17-1. Windows 2000인쇄기특성표

이 그림에서 알수 있는바와 같이 PrintDlgEx( )는 사용자가 인쇄기를 설정하는 특성표를 제공한다. 결국 PrintDlgEx( )를 사용하여 장치상황을 얻는 방법을 사용하면 그와 동시에 사용자가 인쇄설정을 하도록 할수 있다.

PrintDlgEx( )는 강력하면서도 유연한 기능을 제공한다. 먼저 PrintDlgEx( )의 기본적인 사용법을 설명한다.

Windows 2000의 새로운 기능 : PrintDlgEx( )함수는 종전의 PrintDlg( )함수를 치환한것이다. PrintDlgEx()가 보다 강력하며 유연한 기능을 제공해 준다. 그러므로 새로운 응용프로그람들에서는 PrintDlgEx( )를 사용해야 한다.

*PrintDlgEx( )*선언은 다음과 같다.

```
BOOL PrintDlgEx(LPPRINTDLGEX printDlgEx);
```

PrintDlgEx( )는 사용자가 [Print]단추를 눌러(인쇄를 실행하는 경우) 대화칸을 닫으면 S_OK를 돌려 준다. PrintDlgEx()를 사용하려면 COMMDLG.H를 포함시켜야 한다. 이 머리부파일은 보통 WINDOWS.H 를 포함시키면 자동적으로 포함된다. 그러나 리용하는 번역프로그람이 어떤 리유로 COMMDLG.H를 자동적으로 포함하지 않는 경우에는 이 지정을 추가하여야 한다.

역시 printDlgEx()는 새로운 함수이므로 프로그람코드의 선두에 *WINVER*마크로에 0x0500을 정의하여 머리부파일의 적절한 정보가 포함되도록 해야 한다.

PrintDlgEx 파라메터가 가리키는 PRINTDLGEX 구조체의 내용은 PrintDlgEx( )함수의 조작을 설정하는것이다. *PRINTDLGEX*구조체의 정의를 아래에 보여 주었다.

```
typedef struct tagPDEX {
    DWORD lStructSize;
    HWND hwndOwner;
    HGLOBAL hDevMode;
    HGLOBAL hDevName;
    HDC hDC;
    DWORD Flags;
    DWORD Flags2;
    DWORD ExclusionFlags;
    DWORD nPageRanges;
    DWORD nMaxPageRanges;
    LPPRINTPAGERANGE lpPageRanges;
    DWORD nMinPage;
    DWORD nMaxPage;
    DWORD nCopies;
    HINSTANCE hInstance;
    LPCSTR lpPrintTemplateName;
    LPUNKNOWN lpCallback;
    DWORD nPropertyPages;
    HPROPSHEETPAGE *lphPropertyPages;
    DWORD nStartPage;
    DWORD dwResultAction;
}    PRINTDLGEX;
```

lStructSize 에 *PRINTDLGEX* 구조체의 크기를 설정한다. hwndOwner 에 PrintDlgEx( )의 어미창문의 손잡이를 설정한다. hDevMode 에는 대역변수인 *DEVMODE* 구조체의 손잡이를 설정한다. 이 파라메터에는 함수를 호출하기전에는 대화칸의 초기화정보가 설정되며 함수를 호출한후에는 매 조종체들의 상태가 보관된다.

hDevMode 에 NULL 을 설정할수도 있다. 이 경우에는 PrintDlgEx( )가 DEVMODE 구조체의 작성과 초기화를 진행하고 그의 손잡이를 hDevMode 성원에 돌려준다. DEVMODE 구조체는 이 장의 실례프로그람에서는 리용하지 않는다.

hDevNames 에는 대역변수인 DEVNAMES 구조체의 손잡이를 설정한다. 이 구조체

는 인쇄기구동프로그람의 이름, 인쇄기의 이름 및 인쇄기포구의 이름을 정의한다. 이 이름들은 PrintDlgEx( )의 대화칸을 초기화하는데 사용된다. 함수를 호출한후에는 이 성원들에 사용자에 의하여 선택된 이름들이 설정된다. hDevNames 에 NULL 을 설정할수도 있다. 이 경우에는 PrintDlgEx( )가 *DEVNAMES* 구조체의 작성과 초기화를 진행하며 그의 손잡이를 hDevNames 에 돌려 준다.

DEVNAMES 구조체는 이 장의 실례프로그람에서는 쓰이지 않는다. hDevNames 와 hDevMode 에 모두 령을 설정하면 체계설정의 인쇄기가 사용된다.

함수를 호출한후에는 Flags 성원에 지정된 값에 따라서 hDC 에 인쇄기장치상황 또는 정보상황의 어느 하나가 돌려 진다. 이 장에서는 hDC 에 인쇄기장치상황이 돌려 지게 하였다.(*정보상황*이란 장치상황의 작성을 진행하지 않고 그 정보만을 보관한것이다.)

nPageRanges 와 lpPageRanges 를 사용하여 페지범위를 설정한다. 돌림값으로서 nPageRanges 에는 사용자에 의해 선택된 페지범위가 보관된다. 한페지이상의 범위를 선택할수 있다. 만일 대화칸에서 페지범위의 설정을 비표시상태로 하려면 Flags 성원에 PD_NOPAGENUMS 를 포함시킨다.

nMaxPageRanges 에 lpPageRanges 가 가리키는 배렬의 길이를 설정한다. Flags 에 PD_NOPAGENUMS 가 포함되여 있는 경우에는 이 파라메터가 무시된다.

lpPageRanges 에는 *PRINTPAGERANGE 구조체*배렬의 지시자를 설정한다. PRINTPAGERANGE 구조체의 정의를 아래에 표시한다.

```
typedef struct tagPRINTPAGERANGE {
    DWORD nFromPage;        // 개시페지
    DWORD nToPage;          // 완료페지
} PRINTPAGERANGE;
```

Flags 에 PD_NOPAGENUMS 가 포함되여 있는 경우는 lpPageRange 가 무시된다.

nMinPage 에는 대화칸에서 지정할수 있는 최소페지번호를 설정하며 nMaxPage 에는 대화칸에서 지정할수 있는 최대페지번호를 설정한다. Flags 에 PD_NOPAGENUMS 가 포함되여 있는 경우는 이 성원들이 무시된다.

nCopies 에 대화칸의 [Number of copies]라는 편집칸에 표시되는 초기값을 설정한다. 함수를 호출한후는 사용자에 의해 지정된 인쇄부수가 nCopies 에 보관된다. 응용프로그람은 사용자에게 요구되는 인쇄부수에 따라 실제적인 인쇄를 진행한다.

인쇄특성표로 다른 도안(대화칸본보기)을 사용할수도 있다. 그러자면 hInstance 에 대체대화칸본보기의 실체손잡이를 설정하고 Flags 에 PD_ENABLEPRINT TEMPLATEHANDLE 을 포함시킨다. 대체대화칸을 사용하지 않으려는 경우에는 hInstance 에 NULL 을 설정한다.

lpPrintTemplateName 에 대화칸본보기의 자원이름을 설정하여도 대체대화칸을 사용할수 있다. 이 경우에는 hInstance 에 대화칸을 포함한 모듈의 실체손잡이를 설정한다.

Flags 에 PD_ENABLEPRINTTEMPLATE 가 포함되여 있지 않은 경우에는 lpPrintTemplateName 이 무시된다.

lpCallback 에는 역호출정보를 얻기 위한 *역호출함수*의 지시자를 설정한다. 이러한 정보가 필요 없는 경우에는 lpCallback 에 NULL 을 설정한다. 이 장의 실례프로그람에서는 역호출정보를 사용하지 않는다.

nPropertyPages 에는 lphPropertyPages 가 가리키는 배렬의 길이를 설정한다. 이 배렬은 인쇄특성표조종체에 추가되는 보조특성표의 손잡이를 보관하는것이다. 보조특성표가 필요 없는 경우에는(보통 필요 없다.) nPropertyPages 에 령을 설정하며 lphPropertyPages 에는 NULL 을 설정한다.

nStartPage 에는 인쇄특성표가 능동상태로 되였을 때 제일 처음 표시될 페지를 설정한다. 일반적인 프로그람들에서는 이 값을 START_PAGE_GENERAL 로 한다. PrintDlgEx( )의 호출이 성공하면 그의 조작결과가 dwResultAction 에 돌려 진다. dwResultAction 은 아래의 어느 한 값으로 된다.

값	의　　미
PD_RESULT_APPLY	사용자는 [Apply]단추를 눌렀으나 [Print]단추는 누르지 않고 있다. 응용프로그람은 인쇄를 실행하지 않는다.
PD_RESULT_CANCEL	사용자는 [Cancel]단추를 눌렀다. 응용프로그람은 인쇄를 실행하지 않는다.
PD_RESULT_PRINT	사용자는 [Print]단추를 눌렀다. 응용프로그람은 인쇄를 실행한다.

PrintDlgEx( )의 호출이 실패한 경우에는 dwResultAction 에 유효한 값이 보관되지 않는다.

Flags 에는 인쇄특성표의 형태나 능동으로 하려는 조종체의 종류를 설정한다. 돌림값으로서 Flags 에 사용자의 조작내용이 돌려 진다. Flags 에 설정되거나 보관되는 값은 표 17-1 에 표시한 값들의 조합으로 된다.

표 17-1. PRINTDLGEX 구조체의 Flag 성원의 값

Flag	효　　과
PD_ALLPAGES	[All]단일선택단추를 선택상태로 한다.(이것은 체계설정이다.) 돌림값인 경우 [All]단일선택단추가 선택되여 있다는 것을 가리킨다.

PD_COLLATE	[Collate]검사칸을 선택상태로 한다. 돌림값인 경우 [Collate]검사칸이 선택되며 선택된 인쇄기가 부단위의 인쇄를 지원하지 않고 있다는것을 표시 이 경우에는 프로그람에서 부단위의 인쇄를 처리하여야 한다.
PD_CURRENTPAGE	돌림값으로서 [Current page]단일선택단추가 선택되여 있다는것을 표시
PD_DISABLEPRINTTOFILE	[Print to file]검사칸을 무효로 한다.
PD_ENABLEPRINTTEMPLATE	lpPrintTemplateName으로 지정된 대체대화칸본보기를 사용한다.
PD_ENABLEPRINTTEMPLATEHANDLE	hInstance로 지정된 대체대화칸본보기를 사용한다.
PD_EXCLUSIONFLAGS	ExclusionFlags에 자료가 보관되여 있다는것을 표시
PD_HIDEPRINTTOFILE	[Print to file]검사칸을 비표시로 한다.
PD_NOCURRENTPAGE	[Current page]단일선택단추를 비표시로 한다.
PD_NOPAGENUMS	[Pages]단일선택단추를 무효로 한다.
PD_NOSELECTION	[Selection]단일선택단추를 무효로 한다.
PD_NOWARNING	경고통보문을 표시하지 않게 한다.
PD_PAGENUMS	[Pages]단일선택단추를 선택상태로 한다. 돌림값인 경우 [Pages]단일선택단추가 선택되여 있다는것을 표시한다.
PD_PRINTTOFILE	[Print to file]검사칸을 선택상태로 한다. 돌림값인 경우 [Print to file]검사칸이 선택되여 있다는것을 표시
PD_RETURNDC	hDC에 장치상황을 얻는다.
PD_RETURNDEFAULT	돌림값으로서 hDevMode와 hDevNames에 체계설정의 인쇄기가 보관된다. 대화칸은 표시되지 않는다. PrintDlgEx( )를 호출할 때 hDevMode와 hDevNames에 NULL을 설정한다.
PD_RETURNIC	hDC에 정보상황을 얻는다.
PD_SELECTION	[Selection]단일선택단추를 선택상태로 한다. 돌림값의 경우 [Selection]단일선택단추가 선택되여 있다는것을 표시한다.

PD_USEDEVMODECOPIESAND COLLATE PD_USEDEVMODECOPIES	[Number of Copies]돌리개조종체 및 [Collate]검사칸을 인쇄기의 기능에 따라 무효로 한다. 이 기발들을 포함시키지 않은 경우는 인쇄부수가 nCopies 성원에 보관되며 부단위의 인쇄가 요구되였다면 PD_COLLATE 기발이 설정된다.
PD_USELARGETEMPLATE	큰 본보기를 사용한다.

Flags2는 사용할수 없으며 령을 설정하여야 한다. ExclusionFlags는 항목의 중복을 피하기 위하여 사용된다. 이것은 특수한 상황에서만 필요하게 된다. 보통 령을 설정한다. 이미 설명한바와 같이 돌림값으로서 hDC 에 인쇄기장치상황이 돌려 진다. 이것을 사용하면 장치상황을 조작하는 *TextOut( )*나 *BitBlt( )* 등의 함수에서 인쇄기에로의 출력이 진행된다. 응용프로그람에서 인쇄처리가 완료되면 DeleteDC( )를 호출하여 인쇄기장치상황을 삭제한다.

인쇄기함수

인쇄를 할 때는 몇가지 인쇄기함수를 사용해야 한다. 아래에 선언들을 보여 주었다.

```
int EndDoc(HDC hPrDC);
int EndPage(HDC hPrDC);
int StartDoc(HDC hPrDC, CONST DOCINFO *Info);
int StartPage(HDC hPrDC);
```

모든 인쇄기함수에서 hPrDC 에는 인쇄기의 장치상황의 손잡이를 설정한다. 이 함수들은 호출이 성공하면 정의값을 돌려 주며 실패하면 부의값을 돌려 준다.

인쇄를 시작하자면 먼저 *StartDoc( )*를 호출하여야 한다. StartDoc( )는 두가지 처리를 진행한다. 첫번째 처리는 인쇄일감(Print job)을 개시하는것이다. 두번째 처리는 일감 ID 를 돌려 주는것이다.

이 장의 실례프로그람은 인쇄일감 ID 를 필요로 하지 않지만 인쇄와 관련된 기타 함수들중에는 일감 ID 를 필요로 하는것들도 있다. Info 파라메터는 *DOCINFO* 구조체에 대한 지시자이다. *DOCINFO* 구조체의 정의는 다음과 같다.

```
typedef struct _DOCINFO {
    int cbSize;
```

```
        LPCSTR lpszDocName;
        LPCSTR lpszOutput;
        LPCSTR lpszDatatype;
        DWORD fwType;
    } DOCINFO;
```

cbSize 에는 DOCINFO 구조체의 크기를 설정한다. lpszDocName 에는 인쇄일감의 이름을 설정한다. lpszOutput 에는 인쇄출력을 받아 들이는 파일이름을 설정한다. 그러나 hPrDC가 가리키는 인쇄기장치상황에 출력하는 경우에는 lpszOutput에 NULL을 설정하여야 한다. lpszDatatype에는 인쇄일감을 기록하는데 사용되는 자료형을 설정한다. 이 성원에 NULL을 설정할수도 있다. fwType에는 보통 령을 설정한다.

한 페지의 인쇄를 시작할 때마다 *StartPage( )*를 호출하여야 한다. 매 페지의 인쇄가 끝나면 *EndPage( )*를 호출하여야 한다. EndPage( )는 인쇄기에 페지바꾸기처리를 진행하게 한다. 모든 인쇄처리가 끝나면 EndDoc( )를 호출하여야 한다. 그러므로 한개 페지를 인쇄하는 처리순차는 다음과 같다.

```
StartDoc(dc, &info);
StartPage(dc);
// 여기에서 자료를 인쇄한다.
EndPage(dc);
EndDoc(dc);
```

이식과 관련한 요점 : 16bit 의 Windows 3.1 에서는 StartDoc( ), StartPage( ), EndPage( ) 및 EndDoc( )의 기능이 탈출코드에 의해 실현되고 있었다. 탈출코드는 Escape( )함수를 호출하여 보낸다. 낡은 프로그람을 이식하는 경우 Escape( )함수를 호출하는 부분을 해당한 인쇄기함수로 바꾸어야 한다.

간단한 인쇄실례프로그람

Windows 2000에서 인쇄처리를 원만히 진행하려면 아직도 많은 지식을 알아 두어야 한다. 그러나 본문의 인쇄는 앞절에서 설명한 기능들을 사용해도 충분히 실현할수 있다. 그러므로 우선 본문을 인쇄하는 간단한 실례프로그람을 작성해 보기로 한다. 실례 17-1의 프로그람은 인쇄기로 여러행의 본문을 인쇄한다.

실례 17-1. Print 프로그람

```
// 간단한 인쇄실례

// 이 정의가 필요한 번역프로그람도 있다.
#define WINVER 0x0500

#include <windows.h>
#include <cstring>
#include "print.h"

#define NUMLINES 20

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
void PrintInit(PRINTDLGEX *printdlgex, HWND hwnd);

char szWinName[] = "MyWin"; // 창문클라스의 이름

int X = 0, Y = 0; // 현재의 출력위치
int maxX, maxY;   // 화면의 크기

HDC memDC;      // 가상장치손잡이
HBITMAP hBit;  // 비트매프의 손잡이
HBRUSH hBrush; // 붓의 손잡이

PRINTDLGEX printdlgex;
DOCINFO docinfo;

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HACCEL hAccel;
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;
```

```
// 창문클라스를 정의한다.
wcl.cbSize = sizeof(WNDCLASSEX);

wcl.hInstance = hThisInst;      // 실체의 손잡이
wcl.lpszClassName = szWinName; // 창문클라스의 이름
wcl.lpfnWndProc = WindowFunc;  // 창문함수
wcl.style = 0;                 // 체계설정의 형식

wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

wcl.lpszMenuName = "PrintDemoMenu"; // 기본차림표

wcl.cbClsExtra = 0; // 보조기억기령역은 필요 없다.
wcl.cbWndExtra = 0;

// 창문의 배경색을 흰색으로 한다.
wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

// 창문클라스를 등록한다.
if(!RegisterClassEx(&wcl)) return 0;

/* 창문클라스가 등록되였으므로
   창문을 작성할수 있다. */
hwnd = CreateWindow(
  szWinName, // 창문클라스의 이름
  "Using the Printer", // 제목
  WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
  CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 너비는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 높이는 Windows가 결정하게 한다.
  NULL,          // 어미창문은 없다.
  NULL,          // 차림표는 없다.
  hThisInst,     // 실체의 손잡이
  NULL           // 추가파라메터는 없다.
);
```

```
    // 건반가속기를 적재한다.
    hAccel = LoadAccelerators(hThisInst, "PrintDemoMenu");

    // 창문을 표시한다.
    ShowWindow(hwnd, nWinMode);
    UpdateWindow(hwnd);

    // 통보문순환고리를 작성한다.
    while(GetMessage(&msg, NULL, 0, 0))
    {
      if(!TranslateAccelerator(hwnd, hAccel, &msg)) {
        TranslateMessage(&msg); // 건반통보를 변환한다.
        DispatchMessage(&msg);  // Windows 2000에 조종을 넘긴다.
      }
    }

    return msg.wParam;
}

/* 이 함수는 Windows 2000으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받아 들인다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
    HDC hdc;
    PAINTSTRUCT ps;
    int response;
    TEXTMETRIC tm;
    char str[80];
    int i;
    unsigned copies;

    switch(message) {
      case WM_CREATE:
        // 화면의 크기를 얻는다.
        maxX = GetSystemMetrics(SM_CXSCREEN);
```

```
    maxY = GetSystemMetrics(SM_CYSCREEN);

    // 가상창문을 작성한다.
    hdc = GetDC(hwnd);
    memDC = CreateCompatibleDC(hdc);
    hBit = CreateCompatibleBitmap(hdc, maxX, maxY);
    SelectObject(memDC, hBit);
    hBrush = (HBRUSH) GetStockObject(WHITE_BRUSH);
    SelectObject(memDC, hBrush);
    PatBlt(memDC, 0, 0, maxX, maxY, PATCOPY);

    // 본문치수를 얻는다.
    GetTextMetrics(hdc, &tm);

    strcpy(str, "This is displayed in the main window.");
    for(i=0; i<NUMLINES; i++) {
      TextOut(memDC, X, Y, str, strlen(str)); // 기억기에 출력한다.
      TextOut(hdc, X, Y, str, strlen(str)); // 창문에 출력한다.
      // 행바꾸기한다.
      Y = Y + tm.tmHeight + tm.tmExternalLeading;
    }

    ReleaseDC(hwnd, hdc);
    break;
  case WM_COMMAND:
    switch(LOWORD(wParam)) {
      case IDM_TEXT:
        X = Y = 0;

        // PRINTDLGEX 구조체를 초기화한다.
        PrintInit(&printdlgex, hwnd);

        if(PrintDlgEx(&printdlgex) != S_OK) break;
        else if(printdlgex.dwResultAction != PD_RESULT_PRINT) break;

        docinfo.cbSize = sizeof(DOCINFO);
        docinfo.lpszDocName = "Printing text";
        docinfo.lpszOutput = NULL;
```

```
        docinfo.lpszDatatype = NULL;
        docinfo.fwType = 0;

        // 인쇄기의 본문치수를 얻는다.
        GetTextMetrics(printdlgex.hDC, &tm);

        strcpy(str, "This is printed on the printer.");

        StartDoc(printdlgex.hDC, &docinfo);

        for(copies=0; copies < printdlgex.nCopies; copies++) {
          StartPage(printdlgex.hDC);

          for(i=0; i<NUMLINES; i++) {
            TextOut(printdlgex.hDC, X, Y, str, strlen(str));
            // 행바꾸기한다.
            Y = Y + tm.tmHeight + tm.tmExternalLeading;
          }

          EndPage(printdlgex.hDC);
        }

        EndDoc(printdlgex.hDC);
        DeleteDC(printdlgex.hDC);
        break;
      case IDM_EXIT:
        response = MessageBox(hwnd, "Quit the Program?",
                              "Exit", MB_YESNO);
        if(response == IDYES) PostQuitMessage(0);
        break;
      case IDM_HELP:
        MessageBox(hwnd, "Printing Demo", "Help", MB_OK);
        break;
    }
    break;
  case WM_PAINT: // 다시그리기요구를 처리한다.
    hdc = BeginPaint(hwnd, &ps); // 장치상황을 얻는다.
```

```
        BitBlt(hdc, ps.rcPaint.left, ps.rcPaint.top,
               ps.rcPaint.right-ps.rcPaint.left, // 너비
               ps.rcPaint.bottom-ps.rcPaint.top, // 높이
               memDC,
               ps.rcPaint.left, ps.rcPaint.top,
               SRCCOPY);

        EndPaint(hwnd, &ps); // 장치상황을 해제한다.
        break;
      case WM_DESTROY: // 프로그람을 끝낸다.
        DeleteDC(memDC);
        PostQuitMessage(0);
        break;
      default:
        /* 이 switch 문에서 지정된것 이외의 통보문은
           Windows 2000 에 처리를 맏긴다. */
        return DefWindowProc(hwnd, message, wParam, lParam);
    }
    return 0;
}

// PRINTDLGEX 구조체의 초기화
void PrintInit(PRINTDLGEX *printdlgex, HWND hwnd)
{
    printdlgex->lStructSize = sizeof(PRINTDLGEX);
    printdlgex->hwndOwner = hwnd;
    printdlgex->hDevMode = NULL;
    printdlgex->hDevNames = NULL;
    printdlgex->hDC = NULL;
    printdlgex->Flags = PD_RETURNDC | PD_NOSELECTION |
                        PD_NOPAGENUMS | PD_HIDEPRINTTOFILE |
                        PD_COLLATE | PD_NOPAGENUMS;
    printdlgex->Flags2 = 0;
    printdlgex->ExclusionFlags = 0;
    printdlgex->nMinPage = 1;
    printdlgex->nMaxPage = 1;
    printdlgex->nCopies = 1;
    printdlgex->hInstance = NULL;
```

```
    printdlgex->lpCallback = NULL;
    printdlgex->nPropertyPages = 0;
    printdlgex->lphPropertyPages = NULL;
    printdlgex->nStartPage = START_PAGE_GENERAL;
    printdlgex->dwResultAction = 0;

    printdlgex->lpPrintTemplateName = NULL;
}
```

이 프로그람은 다음과 같은 자원파일을 필요로 한다.

```
#include <windows.h>
#include "print.h"

PrintDemoMenu MENU
{
  POPUP "&Printer Demo"
  {
    MENUITEM "Print &Text\tF2", IDM_TEXT
    MENUITEM "E&xit\tCtrl+X", IDM_EXIT
  }
  MENUITEM "&Help", IDM_HELP
}

PrintDemoMenu ACCELERATORS
{
  VK_F2, IDM_TEXT, VIRTKEY
  VK_F1, IDM_HELP, VIRTKEY
  "^X", IDM_EXIT
}
```

머리부파일 PRINT.H 의 내용을 아래에 보여 주었다. 여기에는 이 장의 뒤부분에서 작성하게 되는 두 실례프로그람에서 사용되는 값들도 정의되여 있다.

```
#define IDM_TEXT        100
#define IDM_BITMAP      101
#define IDM_EXIT        102
#define IDM_HELP        103
```

```
#define IDM_WINDOW          104
#define IDM_ENLARGE         105

#define IDD_EB1             200
#define IDD_EB2             201
#define IDD_UD1             202
#define IDD_UD2             202

#define IDD_TEXT1           210
#define IDD_TEXT2           211
```

첫 인쇄실례프로그람의 상세

이 프로그람에서 중요한 개소는 IDM_TEXT 의 case 문에서 인쇄와 관련한 처리를 진행하는 부분이다. 아래에 다시한번 프로그람코드를 보여 주었다. 이 프로그람코드는 사용자가 [Printer Demo]차림표에서 [Print Text]를 선택하였을 때 실행된다.

```
case IDM_TEXT:
        X = Y = 0;

        // PRINTDLGEX 구조체를 초기화한다.
        PrintInit(&printdlgex, hwnd);

        if(PrintDlgEx(&printdlgex) != S_OK) break;
        else if(printdlgex.dwResultAction != PD_RESULT_PRINT) break;

        docinfo.cbSize = sizeof(DOCINFO);
        docinfo.lpszDocName = "Printing text";
        docinfo.lpszOutput = NULL;
        docinfo.lpszDatatype = NULL;
        docinfo.fwType = 0;

        // 본문치수를 얻는다.
        GetTextMetrics(printdlgex.hDC, &tm);

        strcpy(str, "This is printed on the printer.");
```

```
        StartDoc(printdlgex.hDC, &docinfo);

        for(copies=0; copies < printdlgex.nCopies; copies++) {
          StartPage(printdlgex.hDC);

          for(i=0; i<NUMLINES; i++) {
            TextOut(printdlgex.hDC, X, Y, str, strlen(str));
            // 행바꾸기한다.
            Y = Y + tm.tmHeight + tm.tmExternalLeading;
          }

          EndPage(printdlgex.hDC);
        }

        EndDoc(printdlgex.hDC);
        DeleteDC(printdlgex.hDC);
        break;
```

프로그람코드의 처리내용을 차례로 보자. 우선 인쇄기와 창문에서 본문위치를 가리키는 X 와 Y 가 령으로 초기화되고 있다. 다음 PrintInit( )함수를 호출하여 *PRINTDLGEX* 구조체를 초기화한다. [Selection]단일선택단추 및 [Pages]편집칸을 무효로 하고 있는데 대해 주목하여야 한다. [Print to file]검사칸도 비표시상태로 되여 있다. 이 조종체들은 프로그람에서 사용되지 않는다.

다음 PrintDlgEx( )가 실행되고 있다. 돌림값으로서 사용자에 의해 선택된 인쇄기의 장치상황이 printdlgex.hDC 에 돌려 진다. 계속하여 DOCINFO 구조체가 초기화된다.

다음 인쇄기의 장치상황을 파라메터로 하여 GetTextMetrics( )가 호출되고 있다. 여기서는 본문을 인쇄하므로 필요한 복귀,개행처리를 진행하기 위해 본문치수를 얻어야 한다. 이 값들은 *WM_CREATE* 의 case 문에서 얻고 있는 화면상의 본문과는 다르다. 이것은 중요한 점이다.

인쇄기의 장치상황은 특수한것으로서 프로그람에서 사용되는 창문의 장치상황과 공통적인 속성을 가지고 있지 않다.

인쇄처리를 개시하기 위해서 StartDoc( )가 호출된다. 다음 사용자가 요구하는 부수의 인쇄가 순환고리를 사용하여 진행된다. 인쇄부수는 printdlgex 의 nCopies 성원으로부터 얻는다. 매 페지는 다른 용지에 인쇄되므로 인쇄를 할 때마다 StartPage( )가 호출되고 있다.

PrintDlgEx( )를 호출하여 printdlgex.hDC 에 얻은 장치상황이 TextOut( )함수의 출력대상을 가리키는 파라메터로서 사용되고 있는 점에 주목하여야 한다. 인쇄기의 장치상황을 얻으면 그것을 다른 장치상황과 꼭같이 사용할수 있다. 매 페지의 인쇄처리를 끝

낼 때마다 *EndPage( )* 함수가 호출되고 있다. 모든 인쇄처리가 완료되면 *EndDoc( )*를 실행하고 마감으로 인쇄기의 장치상황을 삭제한다.

이 실례프로그람은 단순한것이지만 인쇄기로 문서를 인쇄하는데 필요한 기본적인 기술들을 모두 반영하고 있다. 이 장의 나머지 부분에서는 비트매프의 인쇄, 인쇄중지함수의 추가 및 척도설정방법에 대하여 설명한다. 모든 실례프로그람들에 있어서 인쇄기로 출력을 진행하는 기본적인 수법은 다 같다.

비트매프의 인쇄

Windows 는 도형방식의 조작체계이므로 도형을 인쇄할수 있으면 좋을것이다. 앞절의 실례프로그람에서 보여 준 TextOut( )를 사용하는 본문의 인쇄는 어떤 의미에서 례외적인것이라고 말할수 있다.

Windows 2000 에서 비트매프를 인쇄하는것은 결코 어려운것이 아니다. 그러나 몇가지 주의해야 할 점들이 있다.

우선 비트매프를 인쇄하기전에 선택된 인쇄기가 도형출력기능을 지원하고 있는가를 확인하여야 한다. 그것은 모든 인쇄기에 도형출력기능이 갖추어 져 있는것은 아니기때문이다. 다음 화면에 표시된 화상의 크기 그대로 비트매프를 인쇄하기 위해 출력의 척도를 설정해야 한다. 또한 인쇄기의 장치상황에는 비트매프를 선택할수 없다는 시끄러운 문제도 있다.(비트매프는 기억기장치상황에만 선택할수 있다.)

그러므로 비트매프를 인쇄하자면 비트매프를 호환성 있는 장치상황에 선택하고 그것을 *StretchBlt( )* 등의 함수를 사용하여 인쇄기의 장치상황에 복사하는 절차가 필요하게 된다. 이 절차를 차례로 설명해 보자.

인쇄기의 주사선기능을 확인하기

모든 인쇄기가 비트매프를 인쇄할수 있는것은 아니다. 례하면 본문밖에 인쇄할수 없는 인쇄기도 있다. Windows 의 전문용어에서는 비트매프를 인쇄할수 있는 인쇄기를 *주사선기능*을 가진 인쇄기라고 부른다.

*주사선*이라는 말은 본래 비데오영상장치를 가리키는 말이였다. 그러나 말의 의미가 통일되여 현재에는 주사선기능을 가지고 있는 장치란 비데오영상장치와 마찬가지로 취급되는 장치라는 의미로 된다. 다시말하여 *주사선기능*을 가진 인쇄기란 도형을 출력할수 있는 인쇄기를 말한다.

현재 쓰이고 있는 일반적인 인쇄기들은 주사선기능을 가지고 있다. 그러나 주사선기능을 가지지 못한 인쇄기들도 아직까지 일부 사용되고 있으므로 비트매프를 인쇄하기에 앞서 확인해 볼 필요가 있다. 그를 위해 *GetDeviceCaps( )*함수를 사용한다. 선언은 다음과 같다.

int GetDeviceCaps(HDC hdc, int attribute);

hdc 에는 정보를 얻을 대상으로 되는 장치상황의 손잡이를 설정한다. attribute 의 값은 얻으려는 정보의 종류를 지정한다. 이 함수는 요구하는 정보를 돌려 준다. 얻을수 있는 정보에는 많은 종류가 있지만 대다수의 정보들은 이 장에서 리용되지 않는다.(자체로 *GetDeviceCaps( )*를 리용하여 여러가지 정보를 얻어보는것이 좋다. 장치에 관한 정보를 놀라울 정도로 많이 얻을수 있다.) 인쇄기로 비트매프를 출력할수 있는가를 확인하려면 attribute 에 RASTERCAPS 를 설정한다. 함수의 돌림값으로서 인쇄기에 주사선기능이 갖추어 져 있는가 없는가가 돌려 진다. 이것은 다음의 몇개 값들의 조합으로 된다.

값	의 미
RC_BANDING	인쇄기의 장치상황은 도형의 밴딩지원을 필요로 한다.
RC_BITBLT	인쇄기의 장치상황은 BitBlt( )의 목표로 얻게 된다.
RC_BITMAP64	인쇄기의 장치상황은 64KB 이상의 크기의 비트매프를 처리할수 있다.
RC_DI_BITMAP	인쇄기의 장치상황은 SetDIBits( ) 나 GetDIBits( ) 등에서 사용되는 장치독립비트매프를 제공하고 있다.
RC_DIBTODEV	인쇄기의 장치상황은 SetDIBitsToDevice( )를 제공한다.
RC_FLOODFIL	인쇄기의 장치상황은 색칠하기를 제공한다.
RC_PALETTE	인쇄기의 장치상황은 조색판을 제공한다.
RC_SCALING	인쇄기의 장치상황은 척도기능을 가지고 있다.
RC_STRETCHBLT	인쇄기의 장치상황은 StretchBlt( )의 목표로 얻게 된다.
RC_STRETCHDIB	인쇄기의 장치상황은 StretchDIBits( )의 목표로 얻게 된다.

이 장의 범위내에서는 *RC_BITBLT* 와 *RC_STRETCHBLT* 의 기능만을 확인할수 있으면 충분하다.

척도의 설정

화면에 표시된 비트매프를 본래의 크기대로 인쇄기로 출력하려면 척도를 적당하게 설정하여야 한다. 그러자면 화면과 인쇄기의 해상도를 알아야 할 필요가 제기된다. 해상도를 조사하려면 역시 GetDeviceCaps( )함수를 리용해야 한다. 수평방향의 1inch 당 화

소(pixel)수를 얻으려면 attribute 에 *LOGPIXELSX* 를 설정해야 한다. 수직방향의 1inch 당 화소수를 얻기 위해서는 attribute 에 *LOGPIXELSY* 를 설정해야 한다. 아래에 실례를 보여 주었다.

```
hres = GetDeviceCaps(hdc, LOGPIXELSX);
vres = GetDeviceCaps(hdc, LOGPIXELSY);
```

hdc 에는 장치상황의 손잡이를 설정한다. hres 에 X 축방향의 1inch 당 화소수가 돌려 지고 vres 에 Y 축방향의 1inch 당 화소수가 돌려 진다.

화면의 장치상황과 인쇄기의 장치상황의 해상도를 얻으면 그것들을 변환하기 위한 배률을 계산할수 있다. 이 배률을 StretchBlt( )에 설정하여 비트매프를 출력하면 인쇄기로 화면과 꼭 같은 화상을 얻을수 있다.

StretchBlt( )

*StretchBlt( )*는 비트매프를 복사하는 기능에 있어서는 이 책의 앞부분에서 설명한 *BitBlt( )*와 같다. 다만 복사처리에 있어서 StretchBlt( )는 복사원천의 비트매프를 복사측의 령역에 맞추어 확대하거나 축소할수 있다. 아래에 선언을 보여 주었다.

```
BOOL StretchBlt(HDC hDest, int DestX, int DestY,
                int DestWidth, int DestHeight,
                HDC hSource, int SourceX, int SourceY,
                int SourceWidth, int SourceHeight,
                DWORD dwHow);
```

hDest 에는 복사측의 장치상황의 손잡이를 설정한다. DestX 와 DestY 에는 비트매프를 써넣는 왼쪽 웃모서리의 자리표를 설정한다. DestWidth 와 DestHeight 에는 복사측의 령역의 너비와 높이를 설정한다. hSource 에는 복사원천측의 장치상황의 손잡이를 설정한다. SourceX 와 SourceY 에는 복사원천측의 비트매프의 왼쪽 웃모서리의 자리표를 설정한다. SourceWidth 와 SourceHeight 에는 복사원천측의 비트매프의 너비와 높이를 설정한다.

StretchBlt( )는 복사측의 크기에 맞추어서 복사원천측의 비트매프를 자동적으로 확대하거나 축소한다. 이 기능은 확대나 축소기능을 가지지 않는 BitBlt( )와의 차이이다.

dwHow 의 값은 비트매프의 비트단위의 복사방법을 지적한다. 여기에서는 BitBlt( )의 파라메터에 설정된 값과 같은것이 사용된다. 흔히 사용되는 값들을 아래에 보여 주었다.

dwHow 의 값	효 과
DSTINVERT	복사측의 비트매프를 반전한다.

SRCAND	복사측의 비트매프와 AND 연산을 진행한다.
SRCCOPY	비트매프를 덧쓰기복사한다.
SRCERASE	복사측의 비트매프를 반전한것과 AND 연산을 진행한다.
SRCINVERT	복사측의 비트매프와 XOR 연산을 진행한다.
SRCPAINT	복사측의 비트매프와 OR 연산을 진행한다.

StretchBlt( )는 비트매프의 인쇄에서 중요한 함수이다. 왜냐하면 출력효과의 배률을 설정할수 있기때문이다. StretchBlt( )는 필요에 따라 복사하는 측의 령역에 맞추어서 복사원천측의 비트매프를 확대 또는 축소한다. 복사측의 령역의 크기에 배률을 맞춤으로서 StretchBlt( )를 사용하여 비트매프의 인쇄척도를 설정할수 있다.

만일 척도의 설정이 필요 없다면 BitBlt( )를 사용하여 비트매프를 인쇄할수도 있다. 이 경우에는 인쇄된 비트매프의 화상이 화면에 표시된것과 다를뿐이다. 두가지 수법을 사용한 실례프로그람을 후에 작성한다.

인쇄기와 호환성이 있는 장치상황을 얻기

비트매프의 인쇄에 있어서 비트매프를 선택할수 있는것은 기억기장치상황뿐이라는 문제가 있다. 사소한것이지만 시끄러운 문제이다. 즉 PrintDlgEx( )나 CreateDC( )에서 얻어 진 인쇄기장치상황에 직접 비트매프를 선택할수 없다는것이다.

비트매프를 인쇄하자면 우선 호환성 있는 기억기장치상황을 작성하고 그 기억기장치상황에 비트매프를 선택하며 마감에 그것을 BitBlt( )또는 StretchBlt( )를 사용하여 인쇄기장치상황에 복사하는 절차가 필요하게 된다.

또 하나의 시끄러운 문제가 있다. 그것은 인쇄하려는 비트매프가 인쇄기장치상황과 호환성이 없을 가능성이 있다는것이다. 이러한 경우에는 먼저 인쇄기와의 호환성이 있는 비트매프를 작성한다.

다음 그 비트매프를 인쇄기와의 호환성이 있는 기억기장치상황에 선택하고 인쇄하려는 비트매프를 인쇄기와의 호환성이 있는 비트매프에 복사한다. 마지막에 그 비트매프를 인쇄기의 장치상황에 복사한다.

이러한 절차는 시끄러운것이지만 Windows 2000 자체가 이렇게 설계되여 있기때문에 다른 방도가 없다. 다만 실제의 프로그람코드는 그닥 복잡한것으로 되지는 않는다.

비트매프를 인쇄하는 실례프로그람

실례 17-2 의 프로그람은 첫 인쇄실례프로그람에 두가지 기능을 추가한것이다. 첫번째 기능은 비트매프를 인쇄하는 기능이다. 척도설정을 진행하는 경우와 진행하지 않는

경우의 인쇄결과의 차이를 확인할수 있게 되여 있다.

두번째 기능은 프로그람의 기본창문의 내용을 그대로 인쇄하는 기능이다. 이것은 간단히 실현할수 있다. 왜냐하면 이 프로그람에서는 제 7장에서 설명한 *가상창문*기술을 사용하므로 기본창문의 내용이 항상 비트매프로서 보관되여 있기때문이다. 그러므로 기본창문의 내용을 인쇄하는것은 비트매프를 인쇄하는 일반적인 순서와 꼭같이 실현할수 있다.

실례 17-2. Print2 프로그람

```
// 비트매프를 인쇄하는 실례프로그람

// 이 정의를 필요로 하는 번역프로그람도 있다.
#define WINVER 0x0500

#include <windows.h>
#include <cstring>
#include "print.h"

#define NUMLINES 25

#define BMPWIDTH 256
#define BMPHEIGHT 128

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
void PrintInit(PRINTDLGEX *printdlgex, HWND hwnd);

char szWinName[] = "MyWin"; // 창문클라스의 이름

int X = 0, Y = 0; // 현재의 출력위치
int maxX, maxY;   // 화면의 크기

HDC memDC, memPrDC; // 가상장치의 손잡이
HBITMAP hBit, hBit2, hImage; // 비트매프의 손잡이
HBRUSH hBrush;      // 붓의 손잡이

PRINTDLGEX printdlgex;
DOCINFO docinfo;
```

```
int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HACCEL hAccel;
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;

  // 창문클라스를 정의한다.
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst;      // 실체의 손잡이
  wcl.lpszClassName = szWinName; // 창문클라스의 이름
  wcl.lpfnWndProc = WindowFunc;  // 창문함수
  wcl.style = 0;                 // 체계설정의 형식

  wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
  wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
  wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

  wcl.lpszMenuName = "PrintDemoMenu2"; // 기본차림표

  wcl.cbClsExtra = 0; // 보조기억기령역은 필요 없다.
  wcl.cbWndExtra = 0;

  // 창문의 배경색을 흰색으로 한다.
  wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

  // 창문클라스를 등록한다.
  if(!RegisterClassEx(&wcl)) return 0;

  /* 창문클라스가 등록되였으므로
     창문을 작성할수 있다. */
  hwnd = CreateWindow(
    szWinName, // 창문클라스의 이름
    "Using the Printer", // 제목
    WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
    CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
```

```
    CW_USEDEFAULT, // Y 자리표는 Windows 가 결정하게 한다.
    CW_USEDEFAULT, // 너비는 Windows 가 결정하게 한다.
    CW_USEDEFAULT, // 높이는 Windows 가 결정하게 한다.
    NULL,          // 어미창문은 없다.
    NULL,          // 차림표는 없다.
    hThisInst,     // 실체의 손잡이
    NULL           // 추가파라메터는 없다.
  );

  // 건반가속기를 적재한다.
  hAccel = LoadAccelerators(hThisInst, "PrintDemoMenu2");

  // 비트매프를 적재한다.
  hImage = LoadBitmap(hThisInst, "MyBP1");

  // 창문을 표시한다.
  ShowWindow(hwnd, nWinMode);
  UpdateWindow(hwnd);

  // 통보문순환고리를 작성한다.
  while(GetMessage(&msg, NULL, 0, 0))
  {
    if(!TranslateAccelerator(hwnd, hAccel, &msg)) {
      TranslateMessage(&msg); // 건반통보를 변환한다.
      DispatchMessage(&msg);  // Windows 2000 에 조종을 넘긴다.
    }
  }

  return msg.wParam;
}

/* 이 함수는 Windows 2000 으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받아 들인다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  HDC hdc;
```

```
PAINTSTRUCT ps;
int response;
TEXTMETRIC tm;
char str[250];
int i;
unsigned copies;
double VidXPPI, VidYPPI, PrXPPI, PrYPPI;
double Xratio, Yratio;
RECT r;

switch(message) {
  case WM_CREATE:
    // 화면의 크기를 얻는다.
    maxX = GetSystemMetrics(SM_CXSCREEN);
    maxY = GetSystemMetrics(SM_CYSCREEN);

    // 가상창문을 작성한다.
    hdc = GetDC(hwnd);
    memDC = CreateCompatibleDC(hdc);
    hBit = CreateCompatibleBitmap(hdc, maxX, maxY);
    SelectObject(memDC, hBit);
    hBrush = (HBRUSH) GetStockObject(WHITE_BRUSH);
    SelectObject(memDC, hBrush);
    PatBlt(memDC, 0, 0, maxX, maxY, PATCOPY);

    ReleaseDC(hwnd, hdc);
    break;
  case WM_COMMAND:
    switch(LOWORD(wParam)) {
      case IDM_TEXT: // 본문을 인쇄한다.
        X = Y = 0;

        // PRINTDLGEX 구조체를 초기화한다.
        PrintInit(&printdlgex, hwnd);

        if(PrintDlgEx(&printdlgex) != S_OK) break;
        else if(printdlgex.dwResultAction != PD_RESULT_PRINT) break;
```

```
    docinfo.cbSize = sizeof(DOCINFO);
    docinfo.lpszDocName = "Printing Text";
    docinfo.lpszOutput = NULL;
    docinfo.lpszDatatype = NULL;
    docinfo.fwType = 0;

    // 인쇄기의 본문치수를 얻는다.
    GetTextMetrics(printdlgex.hDC, &tm);

    strcpy(str, "This is printed on the printer.");

    StartDoc(printdlgex.hDC, &docinfo);

    for(copies=0; copies < printdlgex.nCopies; copies++) {
      StartPage(printdlgex.hDC);

      for(i=0; i<NUMLINES; i++) {
        TextOut(printdlgex.hDC, X, Y, str, strlen(str));
        // 개행한다.
        Y = Y + tm.tmHeight + tm.tmExternalLeading;
      }

      EndPage(printdlgex.hDC);
    }

    EndDoc(printdlgex.hDC);
    DeleteDC(printdlgex.hDC);
    break;
  case IDM_BITMAP: // 비트매프를 인쇄한다.
    // PRINTDLGEX 구조체를 초기화한다.
    PrintInit(&printdlgex, hwnd);

    if(PrintDlgEx(&printdlgex) != S_OK) break;
    else if(printdlgex.dwResultAction != PD_RESULT_PRINT) break;

    docinfo.cbSize = sizeof(DOCINFO);
    docinfo.lpszDocName = "Printing bitmaps";
    docinfo.lpszOutput = NULL;
```

```
docinfo.lpszDatatype = NULL;
docinfo.fwType = 0;

if(!(GetDeviceCaps(printdlgex.hDC, RASTERCAPS)
   & (RC_BITBLT | RC_STRETCHBLT))) {
   MessageBox(hwnd, "Cannot Print Raster Images",
              "Error", MB_OK);
   break;
}

// 인쇄기와 호환성 있는 기억기장치상황을 작성한다.
memPrDC = CreateCompatibleDC(printdlgex.hDC);
// 인쇄기장치상황과 호환성 있는 비트매프를 작성한다.
hBit2 = CreateCompatibleBitmap(printdlgex.hDC, maxX, maxY);
SelectObject(memPrDC, hBit2);

// 비트매프를 기억기장치상황에 선택한다.
SelectObject(memDC, hImage);

// 비트매프를 인쇄기와 호환성을 가지는 장치상황에 복사한다.
BitBlt(memPrDC, 0, 0, BMPWIDTH, BMPHEIGHT,
       memDC, 0, 0, SRCCOPY);

// 1inch 당 화소수를 얻는다.
VidXPPI = GetDeviceCaps(memDC, LOGPIXELSX);
VidYPPI = GetDeviceCaps(memDC, LOGPIXELSY);
PrXPPI = GetDeviceCaps(printdlgex.hDC, LOGPIXELSX);
PrYPPI = GetDeviceCaps(printdlgex.hDC, LOGPIXELSY);

// 배률을 얻는다.
Xratio = PrXPPI / VidXPPI;
Yratio = PrYPPI / VidYPPI;

SelectObject(memDC, hBit); // 가상창문에 보관한다.

StartDoc(printdlgex.hDC, &docinfo);

for(copies=0; copies < printdlgex.nCopies; copies++) {
```

```
        StartPage(printdlgex.hDC);

        // 비트매프를 그대로 인쇄기장치상황에 복사한다.
        BitBlt(printdlgex.hDC, 0, 0, BMPWIDTH, BMPHEIGHT,
               memPrDC, 0, 0, SRCCOPY);

        // 비트매프를 적당한 배률로 복사한다.
        StretchBlt(printdlgex.hDC, 0, BMPHEIGHT + 100,
                   (int) (BMPWIDTH*Xratio),
                   (int) (BMPHEIGHT*Yratio),
                   memPrDC, 0, 0,
                   BMPWIDTH, BMPHEIGHT,
                   SRCCOPY);

        EndPage(printdlgex.hDC);
      }

      EndDoc(printdlgex.hDC);
      DeleteDC(memPrDC);
      DeleteDC(printdlgex.hDC);
      break;
    case IDM_WINDOW: // 창문의 내용을 인쇄한다.
      GetClientRect(hwnd, &r);
      hdc = GetDC(hwnd);

      // 창문에 본문을 표시한다.
      GetTextMetrics(hdc, &tm);
      X = Y = 0;
      strcpy(str, "This is displayed in the main window.");
      for(i=0; i<NUMLINES; i++) {
        TextOut(hdc, X, Y, str, strlen(str));
        TextOut(memDC, X, Y, str, strlen(str));
        // 개행한다.
        Y = Y + tm.tmHeight + tm.tmExternalLeading;
      }

      // 창문에 비트매프를 표시한다.
      SelectObject(memDC, hImage);
```

```
BitBlt(hdc, 100, 100, BMPWIDTH, BMPHEIGHT,
        memDC, 0, 0, SRCCOPY);

// 다시그리기요구에 대응하기 위해 창문의 화상을 보관한다.
SelectObject(memDC, hBit);
BitBlt(memDC, 0, 0, r.right, r.bottom, hdc, 0, 0, SRCCOPY);

// 호환성 있는 장치상황을 작성한다.
memPrDC = CreateCompatibleDC(hdc);
// 호환성 있는 비트매프를 작성한다.
hBit2 = CreateCompatibleBitmap(hdc, r.right, r.bottom);
SelectObject(memPrDC, hBit2);

// 인쇄를 위해 창문의 화상을 보관한다.
BitBlt(memPrDC, 0, 0, r.right, r.bottom,
        hdc, 0, 0, SRCCOPY);

// PRINTDLGEX 구조체를 초기화한다.
PrintInit(&printdlgex, hwnd);

if(PrintDlgEx(&printdlgex) != S_OK) break;
else if(printdlgex.dwResultAction != PD_RESULT_PRINT) break;

docinfo.cbSize = sizeof(DOCINFO);
docinfo.lpszDocName = "Printing Window";
docinfo.lpszOutput = NULL;
docinfo.lpszDatatype = NULL;
docinfo.fwType = 0;

// 1inch 당 화소수를 얻는다.
VidXPPI = GetDeviceCaps(memDC, LOGPIXELSX);
VidYPPI = GetDeviceCaps(memDC, LOGPIXELSY);
PrXPPI = GetDeviceCaps(printdlgex.hDC, LOGPIXELSX);
PrYPPI = GetDeviceCaps(printdlgex.hDC, LOGPIXELSY);

// 배률을 얻는다.
Xratio = PrXPPI / VidXPPI;
Yratio = PrYPPI / VidYPPI;
```

```
        if(!(GetDeviceCaps(printdlgex.hDC, RASTERCAPS)
           & RC_STRETCHBLT)) {
           MessageBox(hwnd, "Cannot Print Raster Images",
                      "Error", MB_OK);
           break;
        }


        StartDoc(printdlgex.hDC, &docinfo);

        for(copies=0; copies < printdlgex.nCopies; copies++) {
          StartPage(printdlgex.hDC);

          StretchBlt(printdlgex.hDC, 0, 0,
                     (int) (r.right*Xratio),
                     (int) (r.bottom*Yratio),
                     memPrDC, 0, 0, (int) r.right, (int) r.bottom,
                     SRCCOPY);

          EndPage(printdlgex.hDC);
        }

        EndDoc(printdlgex.hDC);
        DeleteDC(printdlgex.hDC);
        DeleteDC(memPrDC);
        ReleaseDC(hwnd, hdc);
        break;
      case IDM_EXIT:
        response = MessageBox(hwnd, "Quit the Program?",
                              "Exit", MB_YESNO);
        if(response == IDYES) PostQuitMessage(0);
        break;
      case IDM_HELP:
        MessageBox(hwnd, "Printing Demo", "Help", MB_OK);
        break;
    }
    break;
```

```
    case WM_PAINT: // 다시그리기요구를 처리한다.
      hdc = BeginPaint(hwnd, &ps); // 장치상황을 얻는다.

      BitBlt(hdc, ps.rcPaint.left, ps.rcPaint.top,
             ps.rcPaint.right-ps.rcPaint.left, // 너비
             ps.rcPaint.bottom-ps.rcPaint.top, // 높이
             memDC,
             ps.rcPaint.left, ps.rcPaint.top,
             SRCCOPY);

      EndPaint(hwnd, &ps); // 장치상황을 해제한다.
      break;
    case WM_DESTROY: // 프로그람을 끝낸다.
      DeleteDC(memDC);
      PostQuitMessage(0);
      break;
    default:
      /* 이 switch 문에서 지정된것 이외의 통보문은
         Windows 2000 에 처리를 맡긴다. */
      return DefWindowProc(hwnd, message, wParam, lParam);
  }

  return 0;
}

// PRINTDLGEX 구조체의 초기화
void PrintInit(PRINTDLGEX *printdlgex, HWND hwnd)
{
  printdlgex->lStructSize = sizeof(PRINTDLGEX);
  printdlgex->hwndOwner = hwnd;
  printdlgex->hDevMode = NULL;
  printdlgex->hDevNames = NULL;
  printdlgex->hDC = NULL;
  printdlgex->Flags = PD_RETURNDC | PD_NOSELECTION |
                      PD_NOPAGENUMS | PD_HIDEPRINTTOFILE |
                      PD_COLLATE | PD_NOPAGENUMS;
  printdlgex->Flags2 = 0;
  printdlgex->ExclusionFlags = 0;
```

```
    printdlgex->nMinPage = 1;
    printdlgex->nMaxPage = 1;
    printdlgex->nCopies = 1;
    printdlgex->hInstance = NULL;
    printdlgex->lpCallback = NULL;
    printdlgex->nPropertyPages = 0;
    printdlgex->lphPropertyPages = NULL;
    printdlgex->nStartPage = START_PAGE_GENERAL;
    printdlgex->dwResultAction = 0;

    printdlgex->lpPrintTemplateName = NULL;
}
```

이 프로그람은 아래와 같은 자원파일을 필요로 한다.

```
#include <windows.h>
#include "print.h"

MyBP1 BITMAP BP.BMP

PrintDemoMenu2 MENU
{
  POPUP "&Printer Demo"
  {
    MENUITEM "Print &Text\tF2", IDM_TEXT
    MENUITEM "Print &Bitmap\tF3", IDM_BITMAP
    MENUITEM "Print &Window\tF4", IDM_WINDOW
    MENUITEM "E&xit\tCtrl+X", IDM_EXIT
  }
  MENUITEM "&Help", IDM_HELP
}

PrintDemoMenu2 ACCELERATORS
{
  VK_F2, IDM_TEXT, VIRTKEY
  VK_F3, IDM_BITMAP, VIRTKEY
  VK_F4, IDM_WINDOW, VIRTKEY
  VK_F1, IDM_HELP, VIRTKEY
```

```
    "^X", IDM_EXIT
}
```

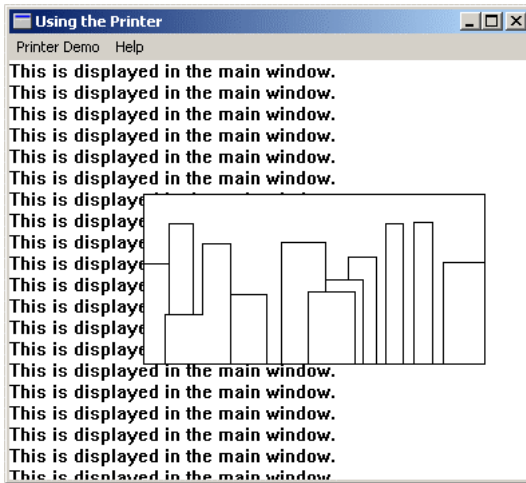
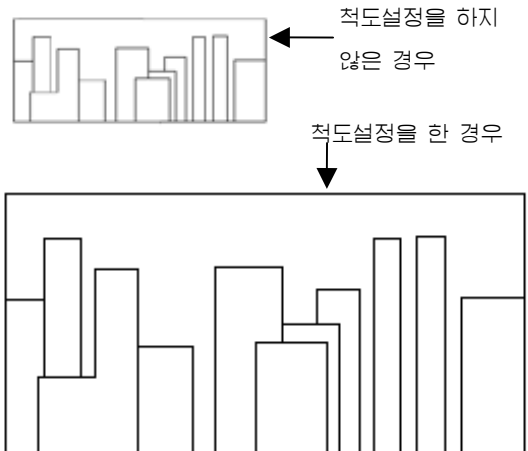
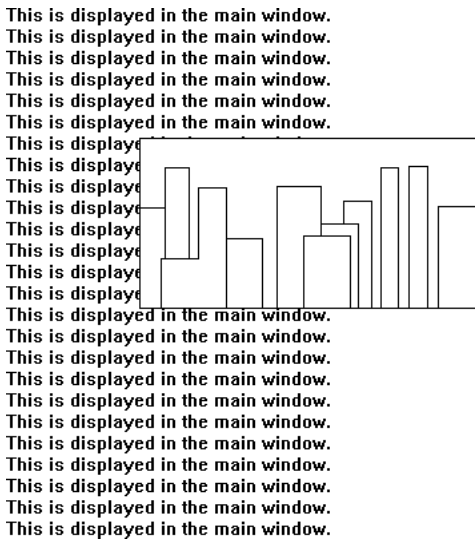
프로그람실행결과를 그림 17-2 에 준다.

ㄱ)

ㄴ)

ㄷ)

그림 17-2. 비트매프를 인쇄하는 실례프로그람의 실행결과

ㄱ) 화면에 표시된 창문, ㄴ) 인쇄된 비트매프, ㄷ) 인쇄결과

이 프로그람은 비트매프파일을 필요로 한다. 프로그람코드에서 볼수 있는것처럼 비트매프의 크기는 너비가 256 화소, 높이 128 화소로 되여야 한다. 그러나 BMPWIDTH 와 BMPHEIGHT 정의를 변경시키면 임의의 크기의 비트매프를 사용할수도 있다. 비트매프파일을 BP.BMP 라는 이름으로 작성하여 둔다.

비트매프를 인쇄하는 실례프로그람의 상세

IDM_BITMAP의 case문 부분의 프로그람코드를 보자. 이 프로그람코드는 사용자가 차림표로부터 [Print Bitmap]를 선택했을 때 실행된다. 아래에 다시 프로그람코드를 보여 준다.

```
case IDM_BITMAP: // 비트매프를 인쇄한다.
        // PRINTDLGEX 구조체를 초기화한다.
        PrintInit(&printdlgex, hwnd);

        if(PrintDlgEx(&printdlgex) != S_OK) break;
        else if(printdlgex.dwResultAction != PD_RESULT_PRINT) break;

        docinfo.cbSize = sizeof(DOCINFO);
        docinfo.lpszDocName = "Printing bitmaps";
        docinfo.lpszOutput = NULL;
        docinfo.lpszDatatype = NULL;
        docinfo.fwType = 0;

        if(!(GetDeviceCaps(printdlgex.hDC, RASTERCAPS)
           & (RC_BITBLT | RC_STRETCHBLT))) {
           MessageBox(hwnd, "Cannot Print Raster Images",
                      "Error", MB_OK);
           break;
        }

        // 인쇄기와 호환성 있는 기억기장치상황을 작성한다.
        memPrDC = CreateCompatibleDC(printdlgex.hDC);
        // 인쇄기장치상황과 호환성 있는 비트매프를 작성한다.
        hBit2 = CreateCompatibleBitmap(printdlgex.hDC, maxX, maxY);
        SelectObject(memPrDC, hBit2);

        // 비트매프를 기억기장치상황에 선택한다.
        SelectObject(memDC, hImage);

        // 비트매프를 인쇄기와 호환성을 가지는 장치상황에 복사한다.
```

```
BitBlt(memPrDC, 0, 0, BMPWIDTH, BMPHEIGHT,
       memDC, 0, 0, SRCCOPY);

   // 1inch 당 화소수를 얻는다.
VidXPPI = GetDeviceCaps(memDC, LOGPIXELSX);
VidYPPI = GetDeviceCaps(memDC, LOGPIXELSY);
PrXPPI = GetDeviceCaps(printdlgex.hDC, LOGPIXELSX);
PrYPPI = GetDeviceCaps(printdlgex.hDC, LOGPIXELSY);

// 배률을 얻는다.
Xratio = PrXPPI / VidXPPI;
Yratio = PrYPPI / VidYPPI;

SelectObject(memDC, hBit); // 가상창문에 보관한다.

StartDoc(printdlgex.hDC, &docinfo);

for(copies=0; copies < printdlgex.nCopies; copies++) {
  StartPage(printdlgex.hDC);

  // 비트매프를 그대로 인쇄기장치상황에 복사한다.
  BitBlt(printdlgex.hDC, 0, 0, BMPWIDTH, BMPHEIGHT,
         memPrDC, 0, 0, SRCCOPY);

  // 비트매프를 적당한 배률로 복사한다.
  StretchBlt(printdlgex.hDC, 0, BMPHEIGHT + 100,
             (int) (BMPWIDTH*Xratio),
             (int) (BMPHEIGHT*Yratio),
             memPrDC, 0, 0,
             BMPWIDTH, BMPHEIGHT,
             SRCCOPY);

  EndPage(printdlgex.hDC);
}

EndDoc(printdlgex.hDC);
DeleteDC(memPrDC);
DeleteDC(printdlgex.hDC);
break;
```

인쇄기의 장치상황을 얻고 docinfo의 초기화가 완료되면 GetDeviceCaps( )를 호출하여 인쇄기가 필요한 주사선기능을 가지고 있는가를 확인한다. 도형출력기능을 가지지 못한 인쇄기라면 비트매프를 인쇄할수 없다.

인쇄기에서 비트매프를 인쇄할수 있는가를 확인하면 인쇄기와 호환성이 있는 장치상황(memPrDC)과 비트매프(hBit2)를 작성한다. 인쇄기와 호환성 있는 비트매프의 크기로서 maxX 와 maxY 를 사용하므로 화면의 크기이내의 임의의 크기의 비트매프를 작성할수 있다.(만일 매우 큰 비트매프를 인쇄하려고 한다면 이 변수들의 값을 크게 할 필요가 있다.)

다음 hBit2를 인쇄기와 호환성 있는 *기억기장치상황*에 설정하고 있다.

표시되는 비트매프(손잡이는 hImage)는 memDC 로 가리키는 기억기장치상황에 선택된다. 이것은 *WM_PAINT* 통보문을 처리하는데 사용되는 가상창문을 지원하기 위한 기억기장치상황과 같은것이다. 여기에서는 두가지 목적으로 이 기억기장치상황이 사용되고 있다.(따로따로 기억기장치상황을 사용할수도 있으나 여기에서는 그럴 필요가 없다.)

다음 비트매프가 memDC로부터 memPrDC에 복사된다. 이 중간처리가 필요한 리유는 BP.BMP 에 보관된 비트매프가 화면의 장치상황과는 호환성이 있어도 인쇄기의 장치상황과는 호환성이 없기때문이다. 그러므로 비트매프를 직접 memPrDC 에 선택할수는 없다.

다음 절차로서 배률계산을 진행한다. 그를 위해 화면과 인쇄기 각각의 1inch 당 화소수를 얻고 그 값으로부터 배률을 구한다.

마지막으로 비트매프를 인쇄기에 출력하고 있다. 먼저 BitBlt( )를 사용하여 그대로 복사하고 있다. 이것에 의하여 두개 장치의 형태상 차이를 고려하지 않고 비트매프가 인쇄된다. 다음 StretchBlt( )를 사용하여 배률을 설정하고 비트매프를 인쇄하고 있다. 인쇄결과를 보면 알수 있는것처럼 척도를 설정한 비트매프는 화면에 표시된 비트매프에 가까운 화상으로 된다.

IDM_WINDOW의 case 문의 프로그람코드는 IDM_BITMAP의 case 문과 류사한 처리를 진행하므로 내용을 쉽게 리해할수 있다.

다시 한보 전진

낡은 형식의 인쇄대화칸의 작성

Windows 2000 이전에는 인쇄공통대화칸을 표시하는데 PrintDlg( )함수가 사용되고 있었다. 이 대화칸에는 최신의 PrintDlgEx( )가 제공하는것과 같은 유연성이 없다. 그러나 낡은 프로그람코드에서 사용되고 있는 PrintDlg( )를 PrintDlgEx( )로 바꾸는 작업을 진행하게 되는 경우도 있게 되므로 PrintDlg( ) 함수의 사용방법을 알아 두어야 할 필요가 있다.

*PrintDlg( )*함수의 선언은 다음과 같다.

```
BOOL PrintDlg(LPPRINTDLG PrintDlg);
```

사용자가 [OK]단추를 눌러서 대화칸을 연 경우 함수의 귀환값으로 령 아닌 값이 돌려 진다. 사용자가 [Cancel]단추(또는 [Esc]건) 혹은 체계차림표를 사용하여 대화칸을 닫은 경우는 령이 돌려 진다.

PrintDlg가 가리키는 PRINTDLG구조체의 내용은 PrintDlg()의 조작방법을 설정하기 위한것이다. *PRINTDLG 구조체*의 정의는 다음과 같다.

```
typedef struct tagPD n{
    DWORD lStructSize;
    HWND hwndOwner;
    HGLOBAL hDevMode;
    HGLOBAL hDevNames;
    HDC hDC;
    DWORD Flags;
    WORD nFromPage;
    WORD nToPage;
    WORD nMinPage;
    WORD nMaxPage;
    WORD nCopies;
    HINSTANCE hInstance;
    LPARAM lCustData;
    LPPRINTHOOKPROC lpfnPrintHook;
    LPSETUPHOOKPROC lpfnSetupHook;
    LPCSTR lpPrintTemplateName;
    LPCSTR lpSetupTemplateName;
    HGLOBAL hPrintTemplate;
    HGLOBAL hSetupTemplate;
}   PRINTDLG;
```

정의를 보면 알수 있는바와 같이 PRINTDLG 구조체의 성원의 대다수는 PRINTDLGEX 구조체의 성원과 같다. 큰 차이는 페지범위를(nFromPage 와 nToPage를 사용하여)한개밖에 선택할수 없다는것이다. PRINTDLGEX구조체에서는 여러개의 페지범위를 지원한다. hSetupTemplate, lcustData 및 lpfnPrintfHook 등의 성원은 PrintDlgEx( )에서는 필요로 하지 않는다.

중지함수의 추가

지금까지의 실례프로그람들에서는 인쇄기(실제로는 인쇄완충기)에 출력을 한 다음 그것을 방치해 두었다. 이것은 출력이 완료되면 프로그람의 일감도 끝난다는것이다. 그러나 실제의 응용프로그람은 이렇게 단순한것이 아니다. 인쇄처리중에 오유가 발생하거나 인쇄일감을 중지하지 않으면 안되는 경우도 있다. 사용자의 결심이 변하여 인쇄일감을 도중에 중지시키려고 하는 경우도 있다.

이러한 상황에 대응하기 위해서는 프로그람에 *인쇄중지함수*와 인쇄가 완료되기전에 인쇄일감을 중지하기 위한 대화칸을 작성하여야 한다. 표준적인 Windows 응용프로그람의 형식을 갖추자면 모든 프로그람이 이러한 기능을 지원할 필요가 있다. 이 절에서는 인쇄를 중지하는 방법을 설명한다.

SetAbortProc( )

중지함수를 작성하기 위해 프로그람에서 *SetAbortProc( )*를 사용한다. 아래에 선언을 보여 주었다.

int SetAbortProc(HDC hPrDC, ABORTPROC AbortFunc);

hPrDC에 인쇄기의 장치상황의 손잡이를 설정한다. AbortFunc에 중지함수로 하려는 함수의 이름을 설정한다. 이 함수는 호출이 성공하면 령보다 큰 값을 돌려 주며 실패하면 SP_ERROR를 돌려 준다.

중지함수의 선언은 다음과 같다.

BOOL CALLBACK AbortFunc(HDC hPrDC, int Code);

이 함수가 호출되면 hPrDC에 인쇄기의 장치상황의 손잡이가 보관된다. 오유가 발생하지 않으면 Code의 값은 령으로 된다. 만일 필요하다면 Code의 값을 확인하여 프로그람에서 적당한 오유처리를 할수도 있다. 그러나 중지함수를 작성해 두면 인쇄관리자(Print manager) 가 오유를 처리하여 주므로 Code를 무시하여도 문제가 없다. 인쇄를 계속하려는 경우는 중지함수의 돌림값으로서 령 아닌 값을 돌려 주며 인쇄를 중지하려는 경우는 령을 돌려 준다.

중지함수안에는 통보문순환고리를 작성하여야 한다. 그러나 통보문을 얻는데는 GetMessage( )가 아니라 PM_REMOVE를 지정한 *PeekMessage( )*를 사용해야 한다. 그 리유는 통보문기다림렬에 통보문이 존재하지 않는 경우에 GetMessage( )는 통보문을 계속 기다리고 있으나 PeekMessage( )는 통보문을 기다리지 않기때문이다. 그러므로 중지함수의 골격코드는 다음과 같이 된다.

```
// 인쇄 중지함수
BOOL CALLBACK AbortFunc(HDC hdc, int err)
{
    MSG message;

    while(PeekMessage(&message, NULL, 0, 0, PM_REMOVE) )  {
            if(!IsDialogMessage(hDlg, &message)) {
                    TranslateMessage(&message);
                    DispatchMessage(&message);
            }
    }
    return printOK; // printOK 는 대역변수이다.
}
```

hDlg 에는 인쇄일감을 중지하는데 사용되는 비양식화대화칸의 손잡이를 설정한다. printOK 는 령 아닌 값으로 초기화된 대역변수이다. 사용자가 인쇄일감을 중지한 경우는 이 변수에 령을 설정한다. 이 처리는 후에 설명하는 비양식화대화칸에서 진행한다.

인쇄를 중지하는 대화칸

중지함수를 작성하면 인쇄일감을 중지시키기 위한 비양식화대화칸을 능동으로 하여야 한다. 이 대화칸에 다양한 기능이나 조종체를 추가할수도 있지만 최소한 인쇄일감을 중지시키기 위한 [Cancel]단추를 포함시켜야 한다.

사용자가 [Cancel]단추를 눌렀을 때 대화칸은 대역변수에 령을 설정한다. 이 대역변수는 앞절에서 설명한 중지함수의 돌림값으로 되여 있는것과 같아야 한다.

완전한 인쇄실례프로그람

실례 17-3 에 보여 준 프로그람은 앞의 실례프로그람에 중지함수를 추가한것이다. 그밖에 또 한가지 기능도 추가되였다. 그것은 확대기능이다. 이 기능을 사용하면 X 축과 Y 축의 배률을 설정할수 있다. 체계설정배률은 1 로 되여 있으므로 확대되지 않는다. 이 값을 1~10 의 범위에서 설정할수 있다. 이 배률을 사용하여 비트매프를 본래의 크기의 최대 10 배까지 확대하여 인쇄할수 있다. 한 방향만으로 확대할수도 있다.

배률은 오르내리기조종체를 사용하여 설정할수 있게 되여 있으므로 이 프로그람에는 COMCTL32.LIB 를 련결하여야 한다.

실례 17-3. Print3 프로그람

```
// 중지함수와 확대기능을 가진 실례프로그람

// 이 정의가 필요한 번역프로그람도 있다.
#define WINVER 0x0500

#include <windows.h>
#include <cstring>
#include <commctrl.h>
#include "print.h"

#define NUMLINES 25
#define SCALEMAX 10

#define BMPWIDTH 256
#define BMPHEIGHT 128

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK EnlargeDialog(HWND, UINT, WPARAM, LPARAM);
void PrintInit(PRINTDLGEX *printdlgex, HWND hwnd);

BOOL CALLBACK AbortFunc(HDC hdc, int err);
LRESULT CALLBACK KillPrint(HWND, UINT, WPARAM, LPARAM);

char szWinName[] = "MyWin"; // 창문클라스의 이름

int X = 0, Y = 0; // 현재 출력위치
int maxX, maxY;   // 화면의 크기

HDC memDC, memPrDC; // 가상장치손잡이
HBITMAP hBit, hBit2, hImage; // 비트매프의 손잡이
HBRUSH hBrush;      // 붓의 손잡이

PRINTDLGEX printdlgex;
DOCINFO docinfo;
```

```
HINSTANCE hInst;

int Xenlarge = 1, Yenlarge = 1;

int printOK = 1;

HWND hDlg = NULL;

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HACCEL hAccel;
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;
  INITCOMMONCONTROLSEX cc;

  // 창문클라스를 정의한다.
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst;      // 실체의 손잡이
  wcl.lpszClassName = szWinName; // 창문클라스의 이름
  wcl.lpfnWndProc = WindowFunc;  // 창문함수
  wcl.style = 0;                 // 체계설정의 형식

  wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
  wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
  wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

  wcl.lpszMenuName = "PrintDemoMenu3"; // 기본차림표

  wcl.cbClsExtra = 0; // 보조기억기령역은 필요 없다.
  wcl.cbWndExtra = 0;

  // 창문의 배경색을 흰색으로 한다.
  wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

  // 창문클라스를 등록한다.
```

```
if(!RegisterClassEx(&wcl)) return 0;

/* 창문클라스가 등록되였으므로
   창문을 작성할수 있다. */
hwnd = CreateWindow(
  szWinName, // 창문클라스의 이름
  "Using the Printer", // 제목
  WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
  CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 너비는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 높이는 Windows가 결정하게 한다.
  NULL,          // 어미창문은 없다.
  NULL,          // 차림표는 없다.
  hThisInst,     // 실체의 손잡이
  NULL           // 추가파라메터는 없다.
);

hInst = hThisInst;

// 건반가속기를 적재한다.
hAccel = LoadAccelerators(hThisInst, "PrintDemoMenu3");

// 비트매프를 적재한다.
hImage = LoadBitmap(hThisInst, "MyBP1");

// 공통조종체를 초기화한다.
cc.dwSize = sizeof(INITCOMMONCONTROLSEX);
cc.dwICC = ICC_UPDOWN_CLASS;
InitCommonControlsEx(&cc);

// 창문을 표시한다.
ShowWindow(hwnd, nWinMode);
UpdateWindow(hwnd);

// 통보문순환고리를 작성한다.
while(GetMessage(&msg, NULL, 0, 0))
{
```

```
    if(!TranslateAccelerator(hwnd, hAccel, &msg)) {
      TranslateMessage(&msg); // 전반통보를 변환한다.
      DispatchMessage(&msg);  // Windows 2000 에 조종을 넘긴다.
    }
  }

  return msg.wParam;
}

/* 이 함수는 Windows 2000 으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받아 들인다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  HDC hdc;
  PAINTSTRUCT ps;
  int response;
  TEXTMETRIC tm;
  char str[250];
  int i;
  unsigned copies;
  double VidXPPI, VidYPPI, PrXPPI, PrYPPI;
  double Xratio, Yratio;
  RECT r;

  switch(message) {
    case WM_CREATE:
      // 화면의 크기를 얻는다.
      maxX = GetSystemMetrics(SM_CXSCREEN);
      maxY = GetSystemMetrics(SM_CYSCREEN);

      // 가상창문을 작성한다.
      hdc = GetDC(hwnd);
      memDC = CreateCompatibleDC(hdc);
      hBit = CreateCompatibleBitmap(hdc, maxX, maxY);
      SelectObject(memDC, hBit);
      hBrush = (HBRUSH) GetStockObject(WHITE_BRUSH);
```

```
        SelectObject(memDC, hBrush);
        PatBlt(memDC, 0, 0, maxX, maxY, PATCOPY);

        ReleaseDC(hwnd, hdc);
        break;
      case WM_COMMAND:
        switch(LOWORD(wParam)) {
          case IDM_ENLARGE:
            DialogBox(hInst, "EnlargeDB", hwnd, (DLGPROC) EnlargeDialog);
            break;
          case IDM_TEXT: // 본문을 인쇄한다.
            X = Y = 0;

            // PRINTDLGEX 구조체를 초기화한다.
            PrintInit(&printdlgex, hwnd);

            if(PrintDlgEx(&printdlgex) != S_OK) break;
            else if(printdlgex.dwResultAction != PD_RESULT_PRINT) break;

            docinfo.cbSize = sizeof(DOCINFO);
            docinfo.lpszDocName = "Printing Text";
            docinfo.lpszOutput = NULL;
            docinfo.lpszDatatype = NULL;
            docinfo.fwType = 0;

            StartDoc(printdlgex.hDC, &docinfo);

            strcpy(str, "This is printed on the printer.");

            // 인쇄기의 본문치수를 얻는다.
            GetTextMetrics(printdlgex.hDC, &tm);

            printOK = 1;
            SetAbortProc(printdlgex.hDC, (ABORTPROC) AbortFunc);
            hDlg = CreateDialog(hInst, "PrCancel", hwnd, (DLGPROC) KillPrint);

            for(copies=0; copies < printdlgex.nCopies; copies++) {
              StartPage(printdlgex.hDC);
```

```
        for(i=0; i<NUMLINES; i++) {
          TextOut(printdlgex.hDC, X, Y, str, strlen(str));
          // 개행한다.
          Y = Y + tm.tmHeight + tm.tmExternalLeading;
        }

        EndPage(printdlgex.hDC);
      }

      if(printOK) {
        DestroyWindow(hDlg);
        EndDoc(printdlgex.hDC);
      }

      DeleteDC(printdlgex.hDC);
      break;
    case IDM_BITMAP: // 비트매프를 인쇄한다.
      // PRINTDLGEX 구조체를 초기화한다.
      PrintInit(&printdlgex, hwnd);

      if(PrintDlgEx(&printdlgex) != S_OK) break;
      else if(printdlgex.dwResultAction != PD_RESULT_PRINT) break;

      docinfo.cbSize = sizeof(DOCINFO);
      docinfo.lpszDocName = "Printing bitmaps";
      docinfo.lpszOutput = NULL;
      docinfo.lpszDatatype = NULL;
      docinfo.fwType = 0;

      if(!(GetDeviceCaps(printdlgex.hDC, RASTERCAPS)
         & (RC_BITBLT | RC_STRETCHBLT))) {
         MessageBox(hwnd, "Cannot Print Raster Images",
                    "Error", MB_OK);
         break;
      }

      // 인쇄기와 호환성 있는 기억기장치상황을 작성한다.
```

```
memPrDC = CreateCompatibleDC(printdlgex.hDC);
// 인쇄기장치상황과 호환성 있는 비트매프를 작성한다.
hBit2 = CreateCompatibleBitmap(printdlgex.hDC, maxX, maxY);
SelectObject(memPrDC, hBit2);

// 비트매프를 기억기장치상황에 선택한다.
SelectObject(memDC, hImage);

// 비트매프를 인쇄기와 호환성 있는 장치상황에 복사한다.
BitBlt(memPrDC, 0, 0, BMPWIDTH, BMPHEIGHT,
       memDC, 0, 0, SRCCOPY);

// 1inch 당 화소수를 얻는다.
VidXPPI = GetDeviceCaps(memDC, LOGPIXELSX);
VidYPPI = GetDeviceCaps(memDC, LOGPIXELSY);
PrXPPI = GetDeviceCaps(printdlgex.hDC, LOGPIXELSX);
PrYPPI = GetDeviceCaps(printdlgex.hDC, LOGPIXELSY);

// 배률을 얻는다.
Xratio = PrXPPI / VidXPPI;
Yratio = PrYPPI / VidYPPI;

SelectObject(memDC, hBit); // 가상창문에 보관한다.

StartDoc(printdlgex.hDC, &docinfo);

printOK = 1;
SetAbortProc(printdlgex.hDC, (ABORTPROC) AbortFunc);
hDlg = CreateDialog(hInst, "PrCancel", hwnd, (DLGPROC) KillPrint);

for(copies=0; copies < printdlgex.nCopies; copies++) {
  StartPage(printdlgex.hDC);

  /* 적절한 배률이 아니라 임의의 배률로
     비트매프를 인쇄기장치상황에 복사한다. */
  StretchBlt(printdlgex.hDC, 0, 0,
             BMPWIDTH * Xenlarge,
             BMPHEIGHT * Yenlarge,
```

```
                    memPrDC, 0, 0, BMPWIDTH, BMPHEIGHT, SRCCOPY);

            // 적절한 배률로 비트매프를 확대한다.
            StretchBlt(printdlgex.hDC, 0, BMPHEIGHT+100*Yenlarge,
                    (int) (BMPWIDTH*Xratio*Xenlarge),
                    (int) (BMPHEIGHT*Yratio*Yenlarge),
                    memPrDC, 0, 0,
                    BMPWIDTH, BMPHEIGHT,
                    SRCCOPY);

            EndPage(printdlgex.hDC);
        }

        if(printOK) DestroyWindow(hDlg);

        EndDoc(printdlgex.hDC);
        DeleteDC(printdlgex.hDC);
        DeleteDC(memPrDC);
        break;
      case IDM_WINDOW: // 창문의 내용을 인쇄한다.
        GetClientRect(hwnd, &r);
        hdc = GetDC(hwnd);

        // 창문에 본문을 표시한다.
        GetTextMetrics(hdc, &tm);
        X = Y = 0;
        strcpy(str, "This is displayed in the main window.");
        for(i=0; i<NUMLINES; i++) {
          TextOut(hdc, X, Y, str, strlen(str));
          TextOut(memDC, X, Y, str, strlen(str));
          // 개행한다.
          Y = Y + tm.tmHeight + tm.tmExternalLeading;
        }

        // 창문에 비트매프를 표시한다.
        SelectObject(memDC, hImage);
        BitBlt(hdc, 100, 100, 256, 128,
              memDC, 0, 0, SRCCOPY);
```

```
// 다시그리기요구에 대처하기 위해 창문의 화상을 보관한다.
SelectObject(memDC, hBit);
BitBlt(memDC, 0, 0, r.right, r.bottom, hdc, 0, 0, SRCCOPY);

// 호환성 있는 장치상황을 작성한다.
memPrDC = CreateCompatibleDC(hdc);
// 호환성 있는 비트매프를 작성한다.
hBit2 = CreateCompatibleBitmap(hdc, r.right, r.bottom);
SelectObject(memPrDC, hBit2);

// 인쇄를 위해 창문의 화상을 보관한다.
BitBlt(memPrDC, 0, 0, r.right, r.bottom,
       hdc, 0, 0, SRCCOPY);

// PRINTDLGEX 구조체를 초기화한다.
PrintInit(&printdlgex, hwnd);

if(PrintDlgEx(&printdlgex) != S_OK) break;
else if(printdlgex.dwResultAction != PD_RESULT_PRINT) break;

docinfo.cbSize = sizeof(DOCINFO);
docinfo.lpszDocName = "Printing Window";
docinfo.lpszOutput = NULL;
docinfo.lpszDatatype = NULL;
docinfo.fwType = 0;

// 1inch 당 화소수를 얻는다.
VidXPPI = GetDeviceCaps(memDC, LOGPIXELSX);
VidYPPI = GetDeviceCaps(memDC, LOGPIXELSY);
PrXPPI = GetDeviceCaps(printdlgex.hDC, LOGPIXELSX);
PrYPPI = GetDeviceCaps(printdlgex.hDC, LOGPIXELSY);

// 배률을 얻는다.
Xratio = PrXPPI / VidXPPI;
Yratio = PrYPPI / VidYPPI;

if(!(GetDeviceCaps(printdlgex.hDC, RASTERCAPS)
```

```
         & RC_STRETCHBLT))
        {
          MessageBox(hwnd, "Cannot Print Raster Images",
                     "Error", MB_OK);
          break;
        }

        StartDoc(printdlgex.hDC, &docinfo);

        printOK = 1;
        SetAbortProc(printdlgex.hDC, (ABORTPROC) AbortFunc);
        hDlg = CreateDialog(hInst, "PrCancel", hwnd, (DLGPROC) KillPrint);

        for(copies=0; copies < printdlgex.nCopies; copies++) {
          StartPage(printdlgex.hDC);

          StretchBlt(printdlgex.hDC, 0, 0,
                     (int) (r.right*Xratio) * Xenlarge,
                     (int) (r.bottom*Yratio) * Yenlarge,
                     memPrDC, 0, 0,
                     (int) r.right, (int) r.bottom,
                     SRCCOPY);

          EndPage(printdlgex.hDC);
        }

        if(printOK) DestroyWindow(hDlg);

        EndDoc(printdlgex.hDC);
        DeleteDC(printdlgex.hDC);
        DeleteDC(memPrDC);
        ReleaseDC(hwnd, hdc);
        break;
      case IDM_EXIT:
        response = MessageBox(hwnd, "Quit the Program?",
                              "Exit", MB_YESNO);
        if(response == IDYES) PostQuitMessage(0);
        break;
```

```
            case IDM_HELP:
              MessageBox(hwnd, "Printing Demo", "Help", MB_OK);
              break;
          }
          break;
        case WM_PAINT: // 다시그리기요구를 처리한다.
          hdc = BeginPaint(hwnd, &ps); // 장치상황을 얻는다.

          BitBlt(hdc, ps.rcPaint.left, ps.rcPaint.top,
                 ps.rcPaint.right-ps.rcPaint.left, // 너비
                 ps.rcPaint.bottom-ps.rcPaint.top, // 높이
                 memDC,
                 ps.rcPaint.left, ps.rcPaint.top,
                 SRCCOPY);

          EndPaint(hwnd, &ps); // 장치상황을 해제한다.
          break;
        case WM_DESTROY: // 프로그람을 끝낸다.
          DeleteDC(memDC);
          PostQuitMessage(0);
          break;
        default:
          /* 이 switch 문에서 지정된것 이외의 통보문은
             Windows 2000 에 처리를 맡긴다. */
          return DefWindowProc(hwnd, message, wParam, lParam);
      }

      return 0;
    }

    // PRINTDLGEX 구조체의 초기화
    void PrintInit(PRINTDLGEX *printdlgex, HWND hwnd)
    {
      printdlgex->lStructSize = sizeof(PRINTDLGEX);
      printdlgex->hwndOwner = hwnd;
      printdlgex->hDevMode = NULL;
      printdlgex->hDevNames = NULL;
      printdlgex->hDC = NULL;
```

```
    printdlgex->Flags = PD_RETURNDC | PD_NOSELECTION |
                        PD_NOPAGENUMS | PD_HIDEPRINTTOFILE |
                        PD_COLLATE | PD_NOPAGENUMS;
    printdlgex->Flags2 = 0;
    printdlgex->ExclusionFlags = 0;
    printdlgex->nMinPage = 1;
    printdlgex->nMaxPage = 1;
    printdlgex->nCopies = 1;
    printdlgex->hInstance = NULL;
    printdlgex->lpCallback = NULL;
    printdlgex->nPropertyPages = 0;
    printdlgex->lphPropertyPages = NULL;
    printdlgex->nStartPage = START_PAGE_GENERAL;
    printdlgex->dwResultAction = 0;

    printdlgex->lpPrintTemplateName = NULL;
}

// 배률을 설정하는 대화함수
BOOL CALLBACK EnlargeDialog(HWND hdwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
    static int tempX=1, tempY=1;

    static long temp;
    static HWND hEboxWnd1, hEboxWnd2;
    static HWND udWnd1, udWnd2;
    int low=1, high=SCALEMAX;


    switch(message) {
      case WM_INITDIALOG:
        hEboxWnd1 = GetDlgItem(hdwnd, IDD_EB1);
        hEboxWnd2 = GetDlgItem(hdwnd, IDD_EB2);
        udWnd1 = CreateUpDownControl(
                      WS_CHILD | WS_BORDER | WS_VISIBLE |
                      UDS_SETBUDDYINT | UDS_ALIGNRIGHT,
                      10, 10, 50, 50,
```

```
                        hdwnd,
                        IDD_UD1,
                        hInst,
                        hEboxWnd1,
                        SCALEMAX, 1, Xenlarge);

        udWnd2 = CreateUpDownControl(
                        WS_CHILD | WS_BORDER | WS_VISIBLE |
                        UDS_SETBUDDYINT | UDS_ALIGNRIGHT,
                        10, 10, 50, 50,
                        hdwnd,
                        IDD_UD2,
                        hInst,
                        hEboxWnd2,
                        SCALEMAX, 1, Yenlarge);

        tempX = Xenlarge;
        tempY = Yenlarge;
        return 1;
      case WM_VSCROLL: // 오르내리기조종체를 처리한다.
        if(udWnd1==(HWND)lParam)
          tempX = GetDlgItemInt(hdwnd, IDD_EB1, NULL, 1);
        else if(udWnd2==(HWND)lParam)
          tempY = GetDlgItemInt(hdwnd, IDD_EB2, NULL, 1);
        return 1;
      case WM_COMMAND:
        switch(LOWORD(wParam)) {
          case IDOK:
            Xenlarge = tempX;
            Yenlarge = tempY;
          case IDCANCEL:
            EndDialog(hdwnd, 0);
            return 1;
          }
        break;
    }

    return 0;
```

```
}

// 인쇄중지함수
BOOL CALLBACK AbortFunc(HDC hdc, int err)
{
  MSG message;

  while(PeekMessage(&message, NULL, 0, 0, PM_REMOVE)) {
    if(!IsDialogMessage(hDlg, &message)) {
      TranslateMessage(&message);
      DispatchMessage(&message);
    }
  }

  return printOK;
}

// 사용자가 인쇄처리를 중지시키게 한다.
LRESULT CALLBACK KillPrint(HWND hdwnd, UINT message,
                           WPARAM wParam, LPARAM lParam)
{
  switch(message) {
    case WM_COMMAND:
      switch(LOWORD(wParam)) {
        case IDCANCEL:
          printOK = 0;
          DestroyWindow(hDlg);
          hDlg = NULL;
          return 1;
        }
      break;
  }

  return 0;
}
```

이 프로그램에서 사용되는 자원파일은 다음과 같다.

```
#include <windows.h>
#include "print.h"

MyBP1 BITMAP BP.BMP

PrintDemoMenu3 MENU
{
  POPUP "&Printer Demo"
  {
    MENUITEM "&Enlarge\tF2", IDM_ENLARGE
    MENUITEM "Print &Text\tF3", IDM_TEXT
    MENUITEM "Print &Bitmap\tF4", IDM_BITMAP
    MENUITEM "Print &Window\tF5", IDM_WINDOW
    MENUITEM "E&xit\tCtrl+X", IDM_EXIT
  }
  MENUITEM "&Help", IDM_HELP
}

PrintDemoMenu3 ACCELERATORS
{
  VK_F2, IDM_ENLARGE, VIRTKEY
  VK_F3, IDM_TEXT, VIRTKEY
  VK_F4, IDM_BITMAP, VIRTKEY
  VK_F5, IDM_WINDOW, VIRTKEY
  "^X", IDM_EXIT
}

EnlargeDB DIALOGEX 10, 10, 97, 77
CAPTION "Enlarge Printer Output"
STYLE WS_POPUP | WS_SYSMENU | WS_VISIBLE
{
  PUSHBUTTON "OK", IDOK, 10, 50, 30, 14
  PUSHBUTTON "Cancel", IDCANCEL, 55, 50, 30, 14
  LTEXT "X Scale Factor", IDD_TEXT1,  15, 1, 25, 20
  LTEXT "Y Scale Factor", IDD_TEXT2,  60, 1, 25, 20
  EDITTEXT IDD_EB1, 15, 20, 20, 12, ES_LEFT |
```

```
            WS_TABSTOP | WS_BORDER
  EDITTEXT IDD_EB2, 60, 20, 20, 12, ES_LEFT |
            WS_TABSTOP | WS_BORDER
}

PrCancel DIALOGEX 10, 10, 100, 40
CAPTION "Printing"
STYLE WS_CAPTION | WS_POPUP | WS_SYSMENU | WS_VISIBLE
{
  PUSHBUTTON "Cancel", IDCANCEL, 35, 12, 30, 14
}
```

프로그람의 실행결과를 그림 17-3 에 준다.

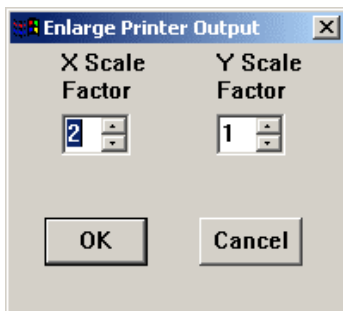
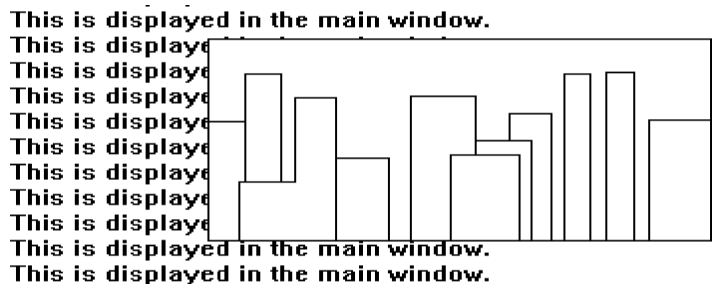



그림 17-3. 최종판 인쇄 실례프로그람의 실행결과

ㄱ - 확대대화칸, ㄴ - X 축을 두배로 확대한 인쇄결과

자체로 해보기

인쇄와 관련하여 더 알아 두어야 할 내용들은 대단히 많다. 이 장을 마치면서 자체로 두가지 시험을 해볼것을 권고한다.

우선 천연색화상을 인쇄해 보는것이다. 만일 흑백인쇄기를 사용하고 있다면 여러가지 색이 어떠한 흑백계조로 변환되는가를 확인해 볼수 있다.

또한 인쇄할 페지범위를 설정해 보아야 한다. 그를 위해서 본문파일의 내용을 인쇄하는 프로그람을 작성해 보는것이 좋을것이다. 사용자가 페지범위를 설정하면 그 범위만을 인쇄한다.

제 18 장

체계자료기지의 리용법과 화면보호기의 작성

이 장에서는 *화면보호기*(Screen Saver)와 체계자료기지(System Registry)라는 두가지 항목을 취급한다. 량자간에 어떤 관계가 있는가 하는 의문을 가질수 있지만 간단한 화면보호기를 작성할 때도 체계자료기지의 기능이 필요하게 된다는것을 알게 될것이다. 왜냐하면 일반적인 화면보호기에서는 설정정보를 체계자료기지에 보관해야 하기때문이다.

이 설정정보는 화면보호기가 기동될 때 참조된다. Windows 2000 은 이러한 정보를 설정하기 위한 적절한 장소로서 *체계자료기지*를 제공하고 있다. 화면보호기는 체계자료기지를 리용하는 응용프로그람으로서 규모가 작다. 이 장에서 화면보호기를 취급하는 리유는 그것이 체계자료기지를 리용하는 좋은 실례로 되기때문이다.

화면보호기는 Windows 2000 응용프로그람으로서는 간단한것이지만 프로그람작성자에게 있어서는 흥미진지한것이기도 하다. 모든 프로그람작성자들이 한번 작성해 보고 싶다고 생각하는 응용프로그람의 하나인것이다.

화면보호기는 변화되지 않는 화면에 대한 보호대책으로서 고안된것이나 오늘에 와서는 그의 역할이 달라졌다고 볼수 있다. 현재 화면보호기의 대다수는 환영을 목적한 통보문, 도형, 기관이나 단체의 명칭 또는 특색 있는 동화상들을 표시하기 위한 목적으로 리용된다.

그에 대응하여 체계자료기지라는 항목은 보다 현실적인것이라고 할수 있으며 Windows 2000 의 프로그람작성기술에서 가장 중요한것의 하나이다. 체계자료기지는 콤퓨터본체와 그 우에서 실행되는 쏘프트웨어의 상태나 설정과 관련한 정보를 보관하는데 리용된다. 체계자료기지의 사용은 생각보다 아주 간단하다.

이 장에서는 두 종류의 간단한 화면보호기를 작성한다. 첫 화면보호기는 아무러한 설정도 할수 없는것이며 체계자료기지도 리용하지 않는다. 이 프로그람의 목적은 모든 화면보호기에 공통되는 기초지식을 주기 위한것이다. 두번째 화면보호기는 일부 설정이 가능한것이며 설정된 정보를 체계자료기지에 보관한다.

미리 말해 두지만 이 두개의 화면보호기들은 다 사람들의 눈을 끄는 요란한 출력을 진행하지는 않는다. 이 프로그람들은 화면보호기의 작성에 필요되는 기술을 주기 위한것일뿐이다. 다만 프로그람작성자들이 앞으로 본격적인 화면보호기를 작성하기 위한 첫 걸음으로 될것이다.

화면보호기의 기초지식

*화면보호기*는 Windows 2000 응용프로그람가운데서 가장 간단한것의 하나일것이다. 왜냐하면 화면보호기는 기본창문을 작성하지 않기때문이다. 화면보호기는 탁상면(화면전체)을 창문으로 사용한다. 또한 화면보호기에서는 WinMain( )함수가 쓰이지 않으며 통보문순환고리도 필요 없다. 화면보호기를 작성하는데 필요한 함수는 세개뿐이다. 이 함수들중 두 함수는 자체내에서 아무러한 처리를 진행하지 않아도 무방하다.

화면보호기를 작성할 때는 SCRNSAVE.H 를 포함시키고 SCRNSAVE.LIB 를 련결하여야 한다. 화면보호기용의 서고는 화면보호기에 필요되는 많은 기능들을 제공해 준다. 그때문에 프로그람에 필요되는 함수가 세개만으로 된다. 화면보호기의 작성에 필요되는 함수들을 상세히 설명해 보자.

화면보호기를 작성하는데 필요되는 세가지 함수

모든 화면보호기에 필요되는 세가지 함수는 다음의 표와 같다.

함　　수	기　　능
ScreenSaverProc( )	화면보호기의 창문수속. 통보문을 접수하고 그에 응답하는 처리를 진행
ScreenSaverConfigureDialog( )	화면보호기설정을 위한 대화함수. 설정이 필요 없으면 함수본체가 비여 있어도 좋다.
RegisterDialogClasses( )	전용클라스를 등록하는데 사용. 전용클라스를 사용하지 않으면 함수본체가 비여 있어도 좋다.

이 함수들의 이름은 Windows 2000 에서 규정되고 있는것이지만 화면보호기의 원천코드안에 함수본체를 써넣어야 한다.(왜냐하면 Win32 에 의해 제공되는 함수가 아니기때문이다.) 매 함수들의 기능을 상세하게 설명해 보자.

*ScreenSaverProc( )*는 *창문수속*이다. 아래에 선언을 보여 주었다.

```
LRESULT WINAPI ScreenSaverProc(HWND hwnd, UINT message,
                WPARAM wParam, LPARAM lParam);
```

이 함수에는 일반적인 창문수속에서와 같은 방법으로 통보문이 전송되여 온다. 다만 중요한 차이점이 하나 있다. 함수가 통보문을 처리하지 않는 경우에는 DefWindowProc( )가 아니라 *DefScreenSaverProc( )*를 호출하는것이다. ScreenSaverProc( )의 hwnd 에 주어지는 창문의 손잡이는 화면전체의 손잡이로 된다. 이것은 탁상면의 손잡이이다. 이 손잡이를 화면보호기가 사용한다.

*ScreenSaverConfigureDialog( )*는 화면보호기의 설정대화칸의 대화함수이다. 아래에 선언을 보여 주었다.

```
BOOL WINAPI ScreenSaverConfigureDialog(HWND hdwnd,
                        UINT message, WPARAM wParam,
                        LPARAM lParam);
```

설정이 필요 없는 화면보호기인 경우 이 함수의 내부에서 진행하여야 할 처리는 령을 돌려 주는것뿐이다. 설정대화칸을 필요로 하는 화면보호기의 경우는 대화칸을 자원파일로서 정의하고 그것에 *DLG_SCRNSAVECONFIGURE* 라는 ID 를 설정하여야 한다. 이 ID 의 값은 SCRNSAVE.H 에 정의되여 있다.

*RegisterDialogClasses( )*는 전용창문클라스를 등록하는데 사용된다. 아래에 선언을 보여 주었다.

```
BOOL WINAPI RegisterDialogClasses(HANDLE hInst);
```

전용창문클라스를 사용하지 않는 화면보호기의 경우에 이 함수의 내부에서 해야 할 처리는 령을 돌려 주는것뿐이다.

이미 설명한것처럼 *ScreenSaverProc( )*에서 처리하지 않는 통보문은 *DefScreenSaverProc( )*에 넘긴다. *DefScreenSaverProc( )*의 선언은 다음과 같다.

```
LRESULT WINAPI DefScreenSaverProc(HWND hwnd, UINT message,
                                  WPARAM wParam, LPARAM lParam);
```

화면보호기를 작성하는데 필요한 두개의 자원

화면보호기에서는 아이콘과 문자렬의 두개 자원을 정의하여야 한다. 아이콘의 ID 는 ID_APP 로 하여야 한다. 이 아이콘은 화면보호기를 식별하기 위한것으로 된다.

문자렬자원의 ID 는 IDS_DESCRIPTION 으로 하여야 한다. 이 문자렬은 화면보호기를 설명하는것이므로 조종판([Property of Screen]조종판)에서 선택가능한 화면보호기의 목록에 표시된다. 문자렬의 길이는 24 문자이하여야 한다.

ID_APP 와 IDS_DESCRIPTION 은 SCRNSAVE.H 에서 정의된다.

문자렬자원은 자원파일내에서 *STRINGTABLE*명령을 포함하여 정의된다. 다음에 일반적인 서식을 준다.

```
STRINGTABLE
{
    ID1 "string"
    ID2 "string"
     ⋮
    Idn "string"
}
```

ID 는 문자렬을 식별하기 위한 식별자이다. 프로그람은 이 ID 를 사용하여 문자렬을 참조한다. 문자렬자원을 적재하기 위해서는 *LoadString( )*을 사용한다. 화면보호기자체는 IDS_DESCRIPTION 로 정의되는 문자렬을 화면보호기를 식별하는데 쓸수 없다. 이 문자렬은 조종판안에서 화면보호기의 설명으로서만 리용된다.

이식과 관련한 요점 : Windows 95/98 에서는 IDS_DESCRIPTION 으로 정의되는 문자렬이 [Property of Screen]의 [Screen Saver]페지표쪽에 표시되지 않는다. 그대신에 화면보호기의 이름이 표시된다. Windows 2000 에서는 여기서 설명한것처럼 IDS_DESCRIPTION 이 정확히 동작한다.

기타 프로그람작성기법

SCRNSAVE.H 에서는 몇가지 대역변수가 선언되고 있다. 그가운데서 이 장을 설명함에 있어서 중요한것은 MainInstance 이다. 이 변수에는 화면보호기의 실체의 손잡이가 보관되여 있다. 이 변수가 필요한것은 화면보호기에서 창문을 작성할 때이다. SCRNSAVE.H 에서 선언되는 다른 대역변수들도 화면보호기를 작성하는데 리용된다.

모든 화면보호기는 시계에 의해 동작한다. 시계에 설정된 시간간격이 경과할 때마다 화면보호기는 *WM_TIMER* 통보문을 받아 들이고 화면의 표시내용을 갱신한다. 화면보호기가 기동할 때 시계를 시동시키며 화면보호기가 완료할 때 시계를 정지시켜야 한다.(시계에 대해서는 제 6 장을 참고)

화면보호기의 프로그람을 번역하면 그 확장자를 EXE 로부터 SCR 로 변경하여야 한다. 확장자를 변경한 화면보호기의 프로그람을 적당한 등록부에 복사한다. 일반사용자들은 화면보호기를 보관하는 등록부를 WINNT＼SYSTEM32 로 하고 있다.(화면보호기를 보관하는 등록부를 확인하기 위한 가장 간단한 방법은 현재 체계에 설치되여 있는 화면보호기가 어느 등록부에 보관되여 있는가를 조사하는것이다.) 화면보호기의 확장자의 변경과 적절한 등록부에로의 복사가 완료되면 조종판을 리용하여 새로운 화면보호기를 선택할수 있다.

최소화면보호기의 작성

간단한 화면보호기를 작성해 보자. 이 화면보호기는 화면우를 이동하는 본문으로 통보문을 표시할뿐이다. 아무러한 설정도 진행할수 없으며 ScreenSaverConfigure Dialog( )함수도 사용할수 없다. 그러나 화면보호기에 필요되는 기본적인 기능은 파악할수 있다.

최소화면보호기를 실례 18-1 에 제시한다.

실례 18-1. Scr 프로그람

```
// 최소화면보호기
#include <windows.h>
#include <scrnsave.h>

// 화면에 표시되는 통보문
char str[80] = "Screen Saver #1";

// 시계의 시간간격
int delay = 200;

// 화면보호기의 창문수속
LRESULT WINAPI ScreenSaverProc(HWND hwnd, UINT message,
                               WPARAM wParam, LPARAM lParam)
{
  static HDC hdc;
  static unsigned int timer;
  static RECT scrdim;
  static SIZE size;
  static int X = 0, Y = 0;
  static HBRUSH hBlkBrush;
  static TEXTMETRIC tm;

  switch(message) {
    case WM_CREATE:
      timer = SetTimer(hwnd, 1, delay, NULL);
      hBlkBrush = (HBRUSH) GetStockObject(BLACK_BRUSH);
      break;
    case WM_ERASEBKGND:
      hdc = GetDC(hwnd);

      // 화면의 크기를 얻는다.
      GetClientRect(hwnd, &scrdim);

      // 화면을 소거한다.
```

```
        SelectObject(hdc, hBlkBrush);
        PatBlt(hdc, 0, 0, scrdim.right, scrdim.bottom, PATCOPY);

        // 문자렬의 너비와 높이를 얻어서 보관한다.
        GetTextMetrics(hdc, &tm);
        GetTextExtentPoint32(hdc, str, strlen(str), &size);

        ReleaseDC(hwnd, hdc);
        break;
      case WM_TIMER:
        hdc = GetDC(hwnd);

        // 앞서 진행한 출력을 소거한다.
        SelectObject(hdc, hBlkBrush);
        PatBlt(hdc, X, Y, X + size.cx, Y + size.cy, PATCOPY);

        // 문자렬을 새로운 위치로 이동한다.
        X += 10; Y += 10;
        if(X > scrdim.right) X = 0;
        if(Y > scrdim.bottom) Y = 0;

        // 문자렬을 표시한다.
        SetBkColor(hdc, RGB(0, 0, 0));
        SetTextColor(hdc, RGB(0, 255, 255));
        TextOut(hdc, X, Y, str, strlen(str));

        ReleaseDC(hwnd, hdc);
        break;
      case WM_DESTROY:
        KillTimer(hwnd, timer);
        break;
      default:
        return DefScreenSaverProc(hwnd, message, wParam, lParam);
    }

  return 0;
}
```

```
// 아무것도 하지 않는 대화함수
BOOL WINAPI ScreenSaverConfigureDialog(HWND hdwnd, UINT message,
                                  WPARAM wParam, LPARAM lParam)
{
  return 0;
}

// 전용클라스를 등록한다.
BOOL WINAPI RegisterDialogClasses(HANDLE hInst)
{
  return 1;
}
```

이 화면보호기는 아래와 같은 자원파일을 필요로 한다.

```
#include <windows.h>
#include <scrnsave.h>

ID_APP ICON SCRICON.ICO

STRINGTABLE
{
  IDS_DESCRIPTION "My Screen Saver #1"
}
```

이 자원파일을 작성하면 화면보호기의 아이콘도 작성하여야 한다. 적당히 설계한 아이콘을 SCRICON.ICO 라는 파일이름으로 보관해 둔다.

최소화면보호기의 상세

화면보호기가 기동하면 두개의 통보문이 발송된다. 최초의 통보문은 *WM_CREATE* 이다. 이 통보문을 받았을 때 시계를 시동시키는것을 비롯한 화면보호기에 필요되는 초기화처리를 진행한다. 이 실례프로그람에서는 시계의 시간간격을 200ms 로 설정하고 있다. 그러므로 시계로부터 화면보호기에 1s 사이에 5 회의 새치기가 걸린다. 이 실례프로그람에서는 화면보호기의 초기화처리로서 검은색 붓을 얻는 조작도 진행되고 있다.

다음으로 화면보호기가 받는 통보문은 *WM_ERASEBKGND* 이다. 이 통보문을 받았을 때 화면보호기는 화면전체를 소거하는 처리를 진행하여야 한다. 물론 화면전체를 소거하기 위한 방법은 한가지만이 아니다. 실례프로그람에서 사용되는 방법은 화면의 크기를 얻고

검은색 붓을 선택하여 그 령역을 *Patblt( )*를 리용하여 검은색으로 색칠하는것이다. hwnd에 탁상면의 손잡이가 전달되므로 GetClientRect( )는 화면전체의 크기를 돌려 준다.

이 화면보호기에 의해 표시되는 통보문의 내용은 고정적인것이므로 WM_ERASEBKGND 를 받았을 때 통보문의 크기를 얻어서 보관하는 처리도 진행되고 있다.

시계에 설정된 시간이 경과할 때마다 WM_TIMER 통보문이 화면보호기에 전송된다. 통보문에 대한 응답으로서 화면보호기는 현재 표시되여 있는 통보문을 소거하고 나서 표시위치를 갱신하여 다시 통보문을 표시한다.

사용자가 건반을 누르든가 마우스를 이동하면 화면보호기는 *WM_DESTROY* 통보문을 접수한다. 화면보호기는 시계를 정지하고 기타 완료처리를 하여야 한다.

이 화면보호기에서는 사용자가 설정할수 있는 항목이나 전용창문클라스가 없으므로 ScreenSaverConfigureDialog( ) 및 *RegisterDialogClasses( )*의 내에서는 아무런 처리도 진행되지 않고 있다.

첫 화면보호기의 문제점

여기서 작성한 화면보호기는 기본적인 기법을 보여 주기 위한것이며 실제로 사용하기에는 그 기능이 불충분하다. 우선 화면에 표시되는 문자렬이 변경될수 없으며 시계의 시간간격도 고정되여 있다. 이것들은 사용자의 의사에 따라 설정될수 있게 되여야 한다.

다행히 이러한 기능들을 화면보호기에 추가하는것은 간단히 해결할수 있다. 다만 그러자면 약간한 문제점이 로출된다. 그것은 어디에 설정정보를 보관하겠는가 하는것이다. 설정가능한 화면보호기로 만들자면 시계의 시간간격과 화면에 표시되는 통보문을 디스크상의 어느 위치에 보관해 두어야 한다.

또한 화면보호기가 기동될 때 이 설정값들을 얻을수 있어야 한다. 이 실례프로그람에서는 설정정보의 자료파일에 보관하는것도 가능하지만 체계자료기지를 사용하는것이 보다 우월한 해결책으로 된다.

체계자료기지의 기초

앞에서 본것처럼 어떤 설정이 가능한 프로그람에서는 설정된 정보를 보관하기 위한 수단이 필요하게 된다.

이전의 콤퓨터환경에서는 매개 프로그람이 설정정보를 보관하기 위한 독자적인 자료파일을 작성하고 있었다. 이러한 파일의 확장자는 CFG 나 DAT 등이였다.(이러한 파일은 점차 사용되지 않고 있지만 아직도 일부 찾아 볼수 있다.) 이러한 개별적인 접근은 DOS 와 같은 조작체계에는 맞는것이였지만 Windows 와 같은 다중과제조작체계에는 맞지 않는다. 왜냐하면 두개이상의 프로그람이 같은 이름의 설정파일을 사용하게 되는 경

우도 있을수 있으며 그렇게 되면 충돌이 발생하기때문이다.

이 문제를 해결하기 위해서 초기의 Windows 에서는 프로그람의 초기화정보파일로서 INI 라는 확장자가 예약되여 있었다. Windows 자체에도 초기화파일이 있으며 WIN.INI 라는 이름으로 되여 있었다. 매개 응용프로그람이 설정정보를 보관하기 위해 WIN.INI 를 사용할수도 있으며 자체의 *INI파일*을 작성하는것도 가능했다.

*INI 파일*은 설정정보를 보관하는 역할을 수행하지만 문제를 완전히 해결하지는 못한다. 그 리유의 하나는 서로 다른 두개의 응용프로그람이 동일한 이름의 INI 파일을 사용하게 된다는 문제가 남아있기때문이다.

콤퓨터에 몇개의 응용프로그람이 설치된다면 대단히 많은 INI 파일이 존재하게 된다. 한 프로그람이 체계에 다른 프로그람이 설치되였는가를 알아 볼 필요가 있는 경우가 있다. 이때 INI 파일을 사용한다고 해서 그것을 쉽게 알아 볼수 있다는 담보로는 되지 못한다.

Microsoft 는 설정파일문제를 해결하기 위한 최종적인 수단으로서 INI 파일과 작별하고 조작체계자료기지(간단히 체계자료기지-System Registry)라는 완전히 새로운 수법을 고안하였다. 체계자료기지의 일부 기능은 Windows 3.1 시대에서부터 지원되고 있었으나 Win32 환경용으로 확장되였다. Windows 2000 에서도 낡은 형식인 INI 파일을 지원하지만 새로 작성하는 응용프로그람에서는 그의 설정정보를 보관하기 위해 체계자료기지를 리용해야 한다.

여러가지 리유로부터 흔히 체계자료기지는 그것의 사용과 리해가 어렵다고 생각하고 있다. 그러나 이것은 오해이며 사실은 정반대이다. 뒤에서 알게 되겠지만 체계자료기지의 사용방법은 매우 간단하다. 몇가지 API 함수의 기능을 파악하면 체계자료기지를 아무런 문제도 없이 사용할수 있다.

이식과 관련한 요점 : 체계자료기지는 Windows 3.1 에서 거의 무시되였다. 대다수의 Windows 3.1 프로그람은 체계자료기지를 사용하지 않는다. 그러나 새로운 응용프로그람 특히 Windows 2000 용으로 작성된 응용프로그람에서는 체계자료기지를 사용해야 한다. 낡은 프로그람을 이식할 때는 체계자료기지를 사용할수 있도록 변경시켜야 한다.

체계자료기지의 구조

*체계자료기지*는 Windows 가 관리하는 특수한 계층구조를 가진 자료기지이며 정보를 사용자, 기계 및 설치된 쏘프트웨어의 세가지로 분류하여 보관한다. 체계자료기지의 거의 모든 부분은 어떤 설정정보를 보관하는데 사용된다. 체계자료기지에 보관된 정보는 2 진수형식으로 되여 있다. 그러므로 체계자료기지의 내용을 변경하거나 참조하는 방법은 두가지밖에 없다. 표준의 체계자료기지편집기인 REGEDIT 또는 *REGEDIT32* 를 리용하든가 체계자료기지관리용의 API 함수를 사용하는것이다. 본문편집기 등으로는 체계자료

기지를 편집할수 없다.

체계자료기지는 나무구조로 된 *열쇠*의 집합이다. 열쇠는 나무의 마디점으로 된다. 열쇠의 이름이 마디점의 이름으로 된다. 열쇠가 비여 있는 경우, 새끼열쇠를 가지는 경우, 값을 가지는 경우가 있다. 이 값은 일반적으로 프로그람의 설정정보를 담고 있게 된다. 그러므로 체계자료기지의 리용은 프로그람에서 열쇠를 작성하고 그 열쇠의 값으로서 설정정보를 보관하는것으로 된다.

프로그람에서 이 설정정보가 필요하게 되였을 때 열쇠를 검색하여 값을 읽어낸다. 사용자가 설정정보를 변경시키는 경우에는 프로그람이 새로운 설정값을 체계자료기지에 써넣는다. 이렇게 체계자료기지의 사용방법은 아주 간단하다.

내장열쇠

체계자료기지에는 *내장열쇠*가 몇종류 있다. 이 열쇠들은 자체의 열쇠에 의한 새끼나무의 뿌리마디점으로 된다. 내장열쇠들의 이름을 아래에 보여 주었다.

열　　쇠	목　　적
HKEY_CLASSES_ROOT	COM에서 사용되는 정보를 보관한다. 파일의 확장자와 응용프로그람들의 련관도 정의한다.
HKEY_CURRENT_CONFIG	현재 하드웨어의 설정정보를 보관한다.
HKEY_CURRENT_USER	현재 사용자와 관련한 정보를 보관한다. 사용자가 사용하는 응용프로그람의 설정정보는 보통 여기에 보관된다.
HKEY_LOCAL_MACHINE	설치되여 있는 쏘프트웨어, 망설정 및 기타 체계준위의 하드웨어 정보 등이 보관되여 있다.
HKEY_USERS	기계를 사용하는 매 사용자의 정보를 보관한다.
HKEY_DYN_DATA	동적성능자료를 보관한다.(Windows 95/98에서만)
HKEY_PERFORMANCE_DATA	동적성능자료를 보관한다.(Windows NT/2000 에서만)

이 열쇠들은 항상 열린 상태로 되여 있으므로 응용프로그람들에서 곧 사용된다. 일반적으로 이 열쇠들가운데서 응용프로그람이 사용하는것은 두개뿐이다. 하나는 기계와 관련한 체계준위의 설정정보가 보관된 *HKEY_LOCAL_MACHINE*이며 다른 한가지는 사용자와 관련한 설정정보가 보관된 *HKEY_CURRENT_USER*이다.

일반적인 설치프로그람들은 HKEY_LOCAL_MACHINE 밑에 열쇠를 작성하고 응용프로그람의 이름, 판번호 및 제작기관의 명칭을 보관한다. 응용프로그람과 관련한 기타 정보가 있으면 HKEY_LOCAL_MACHINE에 보관하기도 한다.

HKEY_CURRENT_USER 밑에는 사용자에 의해 선택된 설정정보가 보관된다. 프로

그람이 설치될 때 이 설정정보의 체계설정값이 보관된다. 응용프로그람은 기동시에 이 설정정보를 참조한다. 화면보호기와 관련한 설정정보도 이 열쇠의 밑에 보관되게 된다.

이 장에서 작성하는 화면보호기는 간단한 실례프로그람이므로 특수한 설치가 요구되지 않는다. 그러므로 HKEY_LOCAL_MACHINE 은 사용하지 않는다. 이 열쇠를 사용하여 체계자료기지로부터 자료를 읽고 쓰는 방법은 HKEY_CURRENT_USER 의 경우와 같다.

내장열쇠밑에는 몇개의 표준적인 새끼열쇠가 있다. 실례로 HKEY_CURRENT_USER 의 밑에는 Software, Control Panel 및 Environment 등의 표준적인 새끼열쇠가 있다.

새로운 프로그람의 설정정보를 체계자료기지에 추가할 때 HKEY_CURRENT_USER 의 새끼열쇠인 Software 의 아래에 추가하는것이 일반적이다. 그러나 화면보호기의 설정정보는 조종판에서 변경하게 되여 있으므로 새끼열쇠인 Control Panel 의 밑에 보관한다. 표준적인 열쇠는 Windows 2000 이 동작하는 거의 모든 콤퓨터에 존재하지만 자동적으로 열리는것은 아니다.

체계자료기지는 계층적인 나무구조로 되여 있으므로 목적하는 열쇠를 완전경로로 지정해야 한다. 열쇠의 경로는 등록부의 경로의 개념과 류사하다. 매개 열쇠는 그의 앞에 \ 기호를 배치하여 구별한다. 실례로 *HKEY_CURRENT_USER* 의 새끼열쇠인 Control Panel 의 밑에 있는 Screensaver 를 참조하는 경우에 다음과 같이 열쇠의 경로를 지정한다.

HKEY_CURRENT_USER \ Control Panel \ Screensaver

열쇠의 이름에는 점(.) 을 포함시켜도 무방하다.

C/C++에서는 문자렬안에서 쓰이는 \기호가 탈출순서(Escape sequence)의 개시를 의미하므로 주의해야 한다. C/C++프로그람에서 \기호를 포함하는 문자렬을 사용하는 경우에는 \기호를 두번 반복하여 쓴다. 실례로 우에서 본 경로를 C/C++의 문자렬로 표시하려면 다음과 같이 서술해야 한다.

"HKEY_CURRENT_USER \\ Control Panel \\ Screensaver"

\기호를 두번 반복하여 쓰지 않으면 체계자료기지용의 함수가 정확히 동작하지 않는다.

체계자료기지에 보관되는 값

체계자료기지의 열쇠에 보관되는 값에는 몇가지 자료형이 있다. 자료를 읽거나 쓸 때는 이름과 자료형을 지정하여야 한다. 표 18-1 에 체계자료기지에서 지원하는 자료형을 준다. 체계자료기지의 열쇠에 값을 추가할 때는 반드시 자료형을 지정한다. 값의 자료형을 자동적으로 판단하는 기능은 체계자료기지에 없다.

표 18-1. 체계자료기지에서 지원하는 자료형

자 료 형	의 미
REG_BINARY	임의의 2 진수자료를 지정하는데 사용되는 일반적인 자료형
REG_DWORD	부호 없는 긴 정수
REG_DWORD_LITTLE_ENDIAN	제일 아래 byte 를 최초에 보관하는 부호 없는 긴 옹근수 이 형식은 Little endian 이라고 한다.
REG_DWORD_BIG_ENDIAN	제일 웃 byte 를 처음에 보관하는 부호 없는 긴 정수 이 형식은 Big endian 이라고 한다.
REG_EXPAND_SZ	전개전의 환경변수를 포함한 문자렬
REG_LINK	유니코드의 기호련결(symbolic link)
REG_MULTI_SZ	문자렬의 배렬. 끝에 두개의 NULL 을 배치하여야 한다.
REG_NONE	정의되지 않은 자료형
REG_QWORD	4 배단어의 값
REG_QWORD_LITTLE_ENDIAN	제일 아래 byte 를 최초에 보관하는 4 배 단어의 값
REG_RESOURCE_LIST	장치구동기의 자원목록
REG_SZ	일반적인 NULL 완료문자렬

열쇠의 작성과 열기

모든 체계자료기지조작은 열쇠를 여는것으로부터 시작한다. 이미 설명한것처럼 내장열쇠들은 항상 열려 져 있다. 그러므로 다른 열쇠를 열 때는 어느 한 내장열쇠를 출발위치로서 지정하게 된다. 이미 존재하는 열쇠를 열자면 *RegOpenKeyEx( )*를 사용한다. 아래에 선언을 보여 주었다.

```
LONG RegOpenKeyEx(HKEY hKey, LPCSTR SubKey,
                  DWORD NotUsed, REGSAM Access,
                  PHKEY Result);
```

hKey 에는 이미 열려 져 있는 열쇠를 설정한다. 이 열쇠는 어느 한 내장열쇠로 될 것이다. 이 열쇠로부터 열기하는 열쇠는 hKey 의 새끼열쇠이여야 한다. 새끼열쇠의 이름은 SubKey 에 설정한다. NotUsed 는 예약된것으로서 령을 설정하여야 한다. Access 는 새끼열쇠의 호출권을 결정한다. 여기에 설정하는 값은 표 18-2 에 표시한 값들의 조합으로 된다. Result 에 새끼열쇠의 손잡이가 돌려 진다. 이 함수는 호출이 성공하면 *ERROR_SUCCESS* 를 돌려 준다. 호출이 실패하면 오유코드를 돌려 준다. 지정된 열쇠가 존재하지 않을 때는 오유로 된다.

표 18-2. 열쇠의 호출권을 가리키는 값

표	의 미
KEY_ALL_ACCESS	모든 호출을 허가
KEY_CREATE_LINK	기호련결(symbolic link)작성을 허가
KEY_CREATE_SUB_KEY	새끼열쇠의 작성을 허가
KEY_ENUMERATE_SUB_KEYS	새끼열쇠의 렬거를 허가
KEY_EXECUTE	읽기를 허가
KEY_NOTIFY	변경에 대한 통지를 허가
KEY_QUERY_VALUE	새끼열쇠의 자료를 읽는것을 허가
KEY_READ	모든 읽기조작을 허가. KEY_ENUMERATE_SUB_KEYS |KEY_NOTIFY|KEY_QUERRY_VALUE 와 같다.
KEY_SET_VALUE	새끼열쇠의 자료의 써넣기를 허가
KEY_WRITE	모든 써넣기조작을 허가한다. KEY_CREATE_SUB_KEY |KEY_SET_VALUE 와 같다.

RegOpenKeyEx( )열쇠를 리용하면 체계자료기지의 열쇠를 열수 있지만 이밖에도 흔히 사용되는 함수로서 *RegCreateKeyEx( )*가 있다. 이 함수는 두가지 목적으로 사용된다. 기존열쇠를 열든가 지정된 열쇠가 존재하지 않는 경우에 새로 열쇠를 작성한다. 아래에 선언을 준다.

```
LONG RegCreatKeyEx(HKEY hKey, LPCSTR SubKey,
                   DWORD NotUsed, LPSTR Class,
                   DWORD How, REGSAM Access,
                   LPSECURITY_ATTRIBUTES SecAttr,
                   PHKEY Result, LPDWORD WhatHappened);
```

hKey 에 열려는 열쇠의 손잡이를 설정한다. SubKey 에는 열거나 작성하려는 새끼열쇠의 이름을 설정한다. NotUsed 는 예약된것으로서 령을 설정하여야 한다. Class 에는 열쇠의 클라스 또는 객체형의 지시자를 설정한다. 이 값은 열쇠를 작성할 때만 필요하게 되며 임의의 문자렬을 설정할수 있다.

REG_OPTION_VOLATILE
REG_OPTION_NON_VOLATILE
REG_OPTION_BACKUP_RESTORE

REG_OPTION_VOLATILE 을 설정한 경우에 열쇠는 일시적인것으로 되며 디스크에 보관되지 않는다. 이 값은 콤퓨터의 전원을 끌 때 없어지게 된다. *REG_OPTION_NON_VOLATILE* 을 지정한 경우 일시적인 열쇠가 아니라 디스크에 보관되는 열쇠가 작성된다. 이 값이 체계설정값이다.

REG_OPTION_BACKUP_RESTORE 를 설정한 경우 예비보관(Backup) 또는 수복(Restore)호출권이 지정된것으로서 열쇠가 작성된다. 이 경우에는 Access 파라메터를 설정할 필요가 없다.

SetAttr 에는 열쇠의 *보안서술자*(security descriptor)를 정의하는 *SECURITY_ATTRIBUTES* 구조체의 지시자를 설정한다. 이 값이 NULL 인 경우는 체계설정의 안전서술자가 사용된다.

Result 에는 작성되거나 열려 진 열쇠의 손잡이가 돌려 진다. WhatHappened 에는 진행된 처리내용이 돌려 진다. 이 값은 새로운 열쇠가 작성되면 *REG_CREATE_NEW_KEY* 로 되며 기존의 열쇠가 열기되면 *REG_OPEN_EXISTING_KEY* 로 된다.

*RegCreateKeyEx( )*는 호출이 성공하면 *ERROR_SUCCESS* 를 돌려 주며 실패하면 오유코드를 돌려 준다.

이식과 관련한 요점 : 체계자료기지와 관련한 함수로서 이름끝에 Ex 가 붙은것들은 Windows 3.1 에서는 지원되지 않는다. RegCreateKeyEx( ) 등은 지원되지 않는다. 그 대신에 낡은 응용프로그람들에서는 RegCreateKey( )가 사용된다. 낡은 응용프로그람을 이식할 때는 낡은 함수들을 이름뒤에 Ex 가 붙은 새 함수들로 변경해야 한다.

값의 보관

열쇠를 열면 거기에 값을 보관할수 있다. 그러자면 *SetRegValueEx( )*를 사용한다. 선언은 다음과 같다.

```
LONG RegSetValueEx(HKEY hKey, LPCSTR Name,
                   DWORD NotUsed, DWORD DataType,
                   CONST LPBYTE lpValue, DWORD SizeOfValue);
```

hKey 에는 *KEY_SET_VALUE* 의 호출권을 지정하여 연 열쇠의 손잡이를 설정한다. Name 에는 값의 이름을 설정한다. 이 이름이 존재하지 않는 경우 새로 열쇠가 추가된다. NotUsed 는 예약된것이며 령을 설정하여야 한다.

DataType 에는 보관하려는 자료의 형을 설정한다. 이 형은 표 18-1 에 표시한 값이여야 한다. lpValue 에는 보관하려는 자료의 지시자를 설정한다. SizeOfValue 에는 보관하려는 자료의 크기를 byte 단위로 설정한다. 문자렬자료의 경우는 문자렬끝기호도 계산된다.

아래의 프로그람은 RegSetValueEx( )를 사용하여 StringTest 의 값으로서 "This is a test"라는 문자렬을 보관하는것이다.

```
strcpy(str, "This is a test");
RegSetValueEx(hRegKey, "StringTest", 0,REG_SZ,
              (LPBYTE) str, strlen(str)+1);
```

문자렬의 크기로서 종결기호(Nail)도 계산하므로 *strlen( )*의 돌림값에 1 을 더해 준다.

RegSetValueEx( )는 호출이 성공하면 *ERROR_SUCCESS* 를 돌려 주며 실패하면 오유코드를 돌려 준다.

값을 얻기

체계자료기지에 값을 보관해 놓으면 임의의 시점에서 프로그람(다른 프로그람도 포함)에서 값을 얻을수 있다. 그를 위해 *RegQueryValueEx( )*함수를 사용한다. 아래에 선언을 보여 주었다.

```
LONG RegQueryValueEx(HKEY hKey, LPCSTR Name,
                     LPDWORD NotUsed, LPDWORD DataType,
                     LPBYTE Value, LPDWORD SizeOfData);
```

hKey 에 *KEY_QUERY_VALUE* 의 호출권을 지정하여 열기한 열쇠의 손잡이를 설정한다. Name 에는 목적하는 값의 이름을 설정한다. 이 값은 지정된 열쇠에 존재하는 값이여야 한다. NotUsed 는 예약끝남이며 NULL 을 설정하여야 한다.

DataType 에는 얻은 값의 자료형이 돌려 진다. 이것은 표 18-1 에 표시된 자료형으로 된다. Value 에는 지정된 값의 자료를 받는 완충기의 지시자를 설정한다. SizeOfData 에는 byte 단위의 완충기의 크기를 보관한 변수의 지시자를 설정한다. 함수를 호출하면 실제로 완충기에 보관된 자료의 크기가 SizeOfData 에 돌려 진다.

아래의 프로그람은 RegQueryValueEx( )를 사용하여 StringTest 의 값으로 되는 문자렬을 얻는 프로그람이다.

```
char str[80];
long size = 80;
RegQueryValueEx(hRegKey, "StringTest", NULL, REG_SZ,
                (LPBYTE) str, &size);
```

RegQueryValueEx( )는 호출이 성공하면 ERROR_SUCCESS 를 돌려 주며 실패하면 오유코드를 돌려 준다.

열쇠의 닫기

열쇠를 닫기 위해서는 *RegCloseKey( )*를 사용한다. 아래에 선언을 보여 주었다.

LONG RegCloseKey(HKEY hKey);

hKey 에는 닫으려는 열쇠의 손잡이를 설정한다. 이 함수는 호출이 성공하면 ERROR_SUCCESS 를 돌려 주며 실패하면 오유코드를 돌려 준다.

다시 한보 전진

기타 체계자료기지용함수

이 장에서 설명하는 체계자료기지용의 함수는 화면보호기를 설정하는데 필요한것뿐이다. 그밖에도 체계자료기지용의 함수가 여러개 있다. 여러분들자신이 사용법을 알아 보시오. 자주 사용되는 함수들을 아래에 보여 주었다.

함　　수	기　　능
RegDeleteKey( )	열쇠를 삭제한다.
RegDeleteValue( )	값을 삭제한다.
RegEnumKeyEx( )	지정된 열쇠의 새끼열쇠를 렬거한다.
RegEnumValue( )	지정된 열쇠의 값을 렬거한다.
RegLoadKey( )	파일로부터 열쇠의 새끼나무를 적재한다.
RegQueryInfoKey( )	지정된 열쇠의 상세정보를 얻는다.
RegSaveKey( )	지정된 열쇠의 새끼열쇠전체를 파일에 보관한다. 값도 보관된다.

체계자료기지에 대해 보다 상세하게 학습하려면 모든 열쇠와 그의 값을 렬거하는 프로그람을 작성해 보는것이 좋다. 체계자료기지를 조사할 때는 함부로 값을 변경하지 않도록 주의하여야 한다. 콤퓨터가 동작하지 않게 될수도 있다.

REGEDIT 의 사용법

체계자료기지용의 API 함수를 리용하면 프로그람에서 체계자료기지를 조작할수 있다. 또한 체계자료기지의 내용을 알아 보거나 변경하려면 *REGEDIT* 또는 REGEDT32(어느것이나 다 체계자료기지편집기라고 부른다.)를 리용할수도 있다. 이 편집기들은 체계자료기지의 열쇠와 값을 표시한다. 열쇠와 값의 추가, 삭제 및 변경도 할수 있다.

일반적으로는 수동적으로 체계자료기지를 변경하지 말아야 한다. 약간한 변경을 하여도 콤퓨터가 동작하지 않게 될수 있기때문이다. 그러나 체계자료기지편집기를 사용하여 체계자료기지의 구조나 내용을 표시하는데 한해서는 완전히 안전하다. 이 프로그람을 리용하여 체계자료기지의 구체적인 구조를 리해할수 있다.

설정가능한 화면보호기의 작성

이 장에서 맨 처음 작성한 간단한 화면보호기를 설정가능한 프로그람으로 개조하려면 세가지 기능을 추가하여야 한다.

첫째 기능은 화면보호기의 자원파일에 설정대화칸을 정의하는것이다. 둘째 기능은 ScreenSaverConfigureDialog( )함수에 처리를 서술하는것이다. 세번째 기능은 설정정보를 체계자료기지에 보관하고 그것을 화면보호기의 기동시에 얻는것이다.

이러한 기능들을 포함한 완전한 프로그람을 실례 18-2 에 보여 주었다. 이 프로그람에서는 *돌리개조종체*를 사용하고 있다. 그러므로 COMCTL32.LIB 를 련결할 필요가 있다.

실례 18-2. Scr2 프로그람

```
// 설정가능한 화면보호기
#include <windows.h>
#include <scrnsave.h>
#include <commctrl.h>
#include <cstring>
#include "scr.h"

#define DELAYMAX 999
#define MSGSIZE 80

// 화면에 표시할 통보문
char str[MSGSIZE+1] = "Screen Saver #2";

// 시계의 시간간격
```

```
long delay;

unsigned long datatype, datasize;
unsigned long result;

// 체계자료기지의 열쇠
HKEY hRegKey;

// 화면보호기의 창문수속
LRESULT WINAPI ScreenSaverProc(HWND hwnd, UINT message,
                               WPARAM wParam, LPARAM lParam)
{
  static HDC hdc;
  static unsigned int timer;
  static RECT scrdim;
  static SIZE size;
  static int X = 0, Y = 0;
  static HBRUSH hBlkBrush;
  static TEXTMETRIC tm;

  switch(message) {
    case WM_CREATE:
      // 화면보호기용의 열쇠를 열거나 작성한다.
      RegCreateKeyEx(HKEY_CURRENT_USER,
         "Control Panel\\Screen Saver.MyScrSaver",
         0, "Screen Saver", 0, KEY_ALL_ACCESS,
         NULL, &hRegKey, &result);

      // 열쇠가 작성된 경우
      if(result==REG_CREATED_NEW_KEY) {
        // 초기값을 보관한다.
        delay = 100;
        RegSetValueEx(hRegKey, "delay", 0,
           REG_DWORD, (LPBYTE) &delay, sizeof(DWORD));
        RegSetValueEx(hRegKey, "message", 0,
           REG_SZ, (LPBYTE) str, strlen(str)+1);
      }
      else { // 열쇠가 이미 존재하는 경우
```

```
        // 시계의 시간간격을 얻는다.
        datasize = sizeof(DWORD);
        RegQueryValueEx(hRegKey, "delay", NULL,
           &datatype, (LPBYTE) &delay, &datasize);

        // 화면에 표시할 통보문을 얻는다.
        datasize = MSGSIZE;
        RegQueryValueEx(hRegKey, "message", NULL,
           &datatype, (LPBYTE) str, &datasize);
      }

      RegCloseKey(hRegKey);

      timer = SetTimer(hwnd, 1, delay, NULL);
      hBlkBrush = (HBRUSH) GetStockObject(BLACK_BRUSH);
      break;
    case WM_ERASEBKGND:
      hdc = GetDC(hwnd);

      // 화면의 크기를 얻는다.
      GetClientRect(hwnd, &scrdim);

      // 화면을 소거한다.
      SelectObject(hdc, hBlkBrush);
      PatBlt(hdc, 0, 0, scrdim.right, scrdim.bottom, PATCOPY);

      // 문자렬의 너비와 높이를 얻어서 보관한다.
      GetTextMetrics(hdc, &tm);
      GetTextExtentPoint32(hdc, str, strlen(str), &size);

      ReleaseDC(hwnd, hdc);
      break;
    case WM_TIMER:
      hdc = GetDC(hwnd);

      // 이전 출력을 소거한다.
      SelectObject(hdc, hBlkBrush);
      PatBlt(hdc, X, Y, X + size.cx, Y + size.cy, PATCOPY);
```

```
        // 문자렬을 새로운 위치로 이동한다.
        X += 10; Y += 10;
        if(X > scrdim.right) X = 0;
        if(Y > scrdim.bottom) Y = 0;

        // 문자렬을 표시한다.
        SetBkColor(hdc, RGB(0, 0, 0));
        SetTextColor(hdc, RGB(0, 255, 255));
        TextOut(hdc, X, Y, str, strlen(str));

        ReleaseDC(hwnd, hdc);
        break;
      case WM_DESTROY:
        KillTimer(hwnd, timer);
        break;
      default:
        return DefScreenSaverProc(hwnd, message, wParam, lParam);
    }

    return 0;
}

// 설정을 진행하기 위한 대화함수
BOOL WINAPI ScreenSaverConfigureDialog(HWND hdwnd, UINT message,
                        WPARAM wParam, LPARAM lParam)
{
    static HWND hEboxWnd;
    static HWND udWnd;

    switch(message) {
      case WM_INITDIALOG:
        // 화면보호기용의 열쇠를 열거나 작성한다.
        RegCreateKeyEx(HKEY_CURRENT_USER,
           "Control Panel\\Screen Saver.MyScrSaver",
           0, "Screen Saver", 0, KEY_ALL_ACCESS,
           NULL, &hRegKey, &result);
```

```
        // 열쇠가 작성된 경우
        if(result==REG_CREATED_NEW_KEY) {
          // 초기값을 얻는다.
          delay = 100;
          RegSetValueEx(hRegKey, "delay", 0,
              REG_DWORD, (LPBYTE) &delay, sizeof(DWORD));
          RegSetValueEx(hRegKey, "message", 0,
              REG_SZ, (LPBYTE) str, strlen(str)+1);
        }
        else { // 열쇠가 이미 존재하는 경우
          // 시계의 시간간격을 얻는다.
          datasize = sizeof(DWORD);
          RegQueryValueEx(hRegKey, "delay", NULL,
              &datatype, (LPBYTE) &delay, &datasize);

          // 화면에 표시할 통보문을 얻는다.
          datasize = MSGSIZE;
          RegQueryValueEx(hRegKey, "message", NULL,
              &datatype, (LPBYTE) str, &datasize);
        }

        // 시계의 시간간격을 설정할 돌리개조종체를 작성한다.
        hEboxWnd = GetDlgItem(hdwnd, IDD_EB1);
        udWnd = CreateUpDownControl(
                      WS_CHILD | WS_BORDER | WS_VISIBLE |
                      UDS_SETBUDDYINT | UDS_ALIGNRIGHT,
                      20, 10, 50, 50,
                      hdwnd,
                      IDD_UPDOWN,
                      hMainInstance,
                      hEboxWnd,
                      DELAYMAX, 1, delay);

        // 현재의 통보문으로 편집칸을 초기화한다.
        SetDlgItemText(hdwnd, IDD_EB2, str);

        return 1;
      case WM_COMMAND:
```

```
      switch(LOWORD(wParam)) {
        case IDOK:
          // 시계의 시간간격을 설정한다.
          delay = GetDlgItemInt(hdwnd, IDD_EB1, NULL, 1);

          // 화면에 표시할 통보문을 얻는다.
          GetDlgItemText(hdwnd, IDD_EB2, str, MSGSIZE);

          // 체계자료기지를 갱신한다.
          RegSetValueEx(hRegKey, "delay", 0,
            REG_DWORD, (LPBYTE) &delay, sizeof(DWORD));
          RegSetValueEx(hRegKey, "message", 0,
             REG_SZ, (LPBYTE) str, strlen(str)+1);

          // 이대로 다음 case 문으로 넘어 간다.
        case IDCANCEL:
          RegCloseKey(hRegKey);
          EndDialog(hdwnd, 0);
          return 1;
        }
      break;
  }
  return 0;
}

// 전용클라스는 등록하지 않는다.
BOOL WINAPI RegisterDialogClasses(HANDLE hInst)
{
  return 1;
}
```

이 프로그람에 필요한 자원파일은 다음과 같다.

```
// 화면보호기용의 대화칸
#include <windows.h>
#include <scrnsave.h>
#include "scr.h"
```

```
ID_APP ICON SCRICON.ICO

STRINGTABLE
{
  IDS_DESCRIPTION "My Screen Saver #2"
}

DLG_SCRNSAVECONFIGURE DIALOGEX 18, 18, 110, 60
CAPTION "Set Screen Saver Options"
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION |
      WS_SYSMENU
{
  PUSHBUTTON "OK", IDOK, 20, 40, 30, 14
  PUSHBUTTON "Cancel", IDCANCEL, 60, 40, 30, 14
  EDITTEXT IDD_EB1, 5, 5, 24, 12, ES_LEFT | WS_TABSTOP | WS_BORDER
  EDITTEXT IDD_EB2, 5, 20, 65, 12, ES_LEFT | WS_TABSTOP |
           WS_BORDER | ES_AUTOHSCROLL
  LTEXT "Delay in milliseconds", IDD_TEXT1, 35, 7, 100, 12
  LTEXT "Message", IDD_TEXT2, 76, 22, 30, 12
}
```

머리부파일 SCR.H의 내용을 아래에 준다.

```
#define IDD_EB1          200
#define IDD_EB2          201

#define IDD_UPDOWN       202

#define IDD_TEXT1        203
#define IDD_TEXT2        204
```

설정가능한 화면보호기의 상세

ScreenSaverConfigureDialog()함수의 내용으로부터 설명을 시작해 보자. 이 대화함수에서는 시계의 시간간격과 화면에 표시되는 통보문의 두가지 항목을 설정할수 있다. 아래에 다시 한번 프로그람코드를 보여 주었다. 이 대화함수가 실행되면 그림 18-1 과 같은 대화칸이 표시된다.

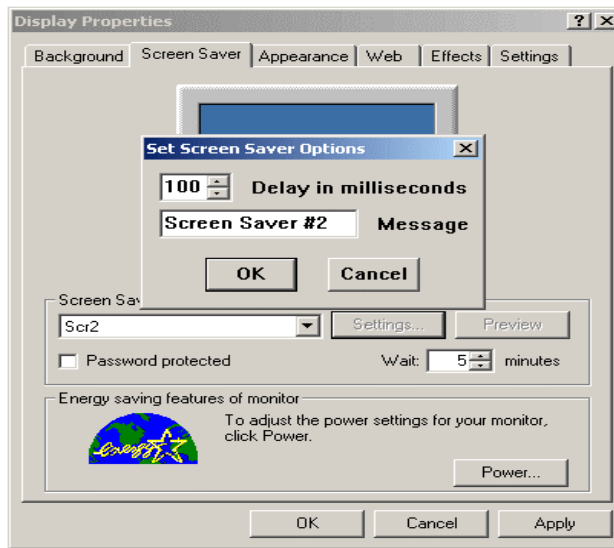



그림 18-1. 화면보호기의 설정대화칸

```
BOOL WINAPI ScreenSaverConfigureDialog(HWND hdwnd, UINT message,
                    WPARAM wParam, LPARAM lParam)
{
  static HWND hEboxWnd;
  static HWND udWnd;

  switch(message) {
    case WM_INITDIALOG:
      // 화면보호기용의 열쇠를 열거나 작성한다.
      RegCreateKeyEx(HKEY_CURRENT_USER,
         "Control Panel\\Screen Saver.MyScrSaver",
         0, "Screen Saver", 0, KEY_ALL_ACCESS,
         NULL, &hRegKey, &result);

      // 열쇠가 작성된 경우
      if(result==REG_CREATED_NEW_KEY) {
        // 초기값을 보관한다.
        delay = 100;
        RegSetValueEx(hRegKey, "delay", 0,
           REG_DWORD, (LPBYTE) &delay, sizeof(DWORD));
```

```
      RegSetValueEx(hRegKey, "message", 0,
          REG_SZ, (LPBYTE) str, strlen(str)+1);
    }
    else { // 열쇠가 이미 존재하는 경우
      // 시계의 시간간격을 얻는다.
      datasize = sizeof(DWORD);
      RegQueryValueEx(hRegKey, "delay", NULL,
          &datatype, (LPBYTE) &delay, &datasize);

      // 화면에 표시할 통보문을 얻는다.
      datasize = MSGSIZE;
      RegQueryValueEx(hRegKey, "message", NULL,
          &datatype, (LPBYTE) str, &datasize);
    }

    // 시계의 시간간격을 설정할 돌리개조종체를 작성한다.
    hEboxWnd = GetDlgItem(hdwnd, IDD_EB1);
    udWnd = CreateUpDownControl(
                  WS_CHILD | WS_BORDER | WS_VISIBLE |
                  UDS_SETBUDDYINT | UDS_ALIGNRIGHT,
                  20, 10, 50, 50,
                  hdwnd,
                  IDD_UPDOWN,
                  hMainInstance,
                  hEboxWnd,
                  DELAYMAX, 1, delay);

    // 현재의 통보문으로 편집칸을 초기화한다.
    SetDlgItemText(hdwnd, IDD_EB2, str);

    return 1;
  case WM_COMMAND:
    switch(LOWORD(wParam)) {
      case IDOK:
        // 시계의 시간간격을 얻는다.
        delay = GetDlgItemInt(hdwnd, IDD_EB1, NULL, 1);

        // 화면에 표시할 통보문을 얻는다.
```

```
            GetDlgItemText(hdwnd, IDD_EB2, str, MSGSIZE);

            // 체계자료기지를 갱신한다.
            RegSetValueEx(hRegKey, "delay", 0,
              REG_DWORD, (LPBYTE) &delay, sizeof(DWORD));
            RegSetValueEx(hRegKey, "message", 0,
               REG_SZ, (LPBYTE) str, strlen(str)+1);

            // 이 상태로 다음 case 문으로 넘어 간다.
          case IDCANCEL:
            RegCloseKey(hRegKey);
            EndDialog(hdwnd, 0);
            return 1;
          }
        break;
    }
    return 0;
  }
```

대화칸이 표시되면 *WM_INITDIALOG* 통보문을 받아 들인다. 이때 대화칸은 ScreenSaver.MyScrSaver 라는 체계자료기지열쇠를 열거나 작성한다. 이 열쇠는 Control Panel 이라는 열쇠의 새끼열쇠로 되여 있으며 Control Panel 은 HKEY_CURRENT_USER 라는 내장열쇠의 새끼열쇠로 되여 있다. 그러므로 열거나 작성하는 열쇠의 경로는 다음과 같이 된다.

HKEY_CURRENT_USER\Control Panel\Screen Saver.MyScrSaver

이미 설명한바와 같이 화면보호기의 설정은 조종판에서 진행하므로 설정자료를 Control Panel 열쇠의 아래에 보관하는것이 일반적이다. 체계자료기지에 열쇠가 존재하지 않는 경우(화면보호기를 처음으로 기동한 때는 존재하지 않는다.)는 *RegCreateKeyEx( )*가 열쇠를 작성한다.

열쇠가 이미 존재하는 경우는 열쇠의 경로가 열기된다. 이 프로그람에 의해 작성된 체계자료기지의 경로를 REGEDIT 에서 표시한것을 그림 18-2 에 보여 주었다. Windows 2000 이 제공하는 화면보호기들도 표시되여 있다.

RegCreateKeyEx( )의 호출이 완료되면 result 의 값을 조사하여 열쇠가 작성된것인가 아니면 이미 있던 열쇠가 열기된것인가를 확인한다. 열쇠가 작성된 경우는 시계의 시간간격과 화면에 표시되는 통보문의 초기값을 보관한다. 그렇지 않은 경우에는 이 값

들을 체계자료기지로부터 읽어 낸다.

초기값을 얻으면 대화칸에 시계의 시간간격을 설정하는 돌리개조종체와 화면에 표시되는 통보문을 설정하는 편집칸을 작성한다. 대화칸에는 두개의 단추도 있다. 사용자가 [OK]단추를 누른 경우 돌리개조종체와 편집칸의 내용을 리용하여 체계자료기지의 내용을 갱신한다.

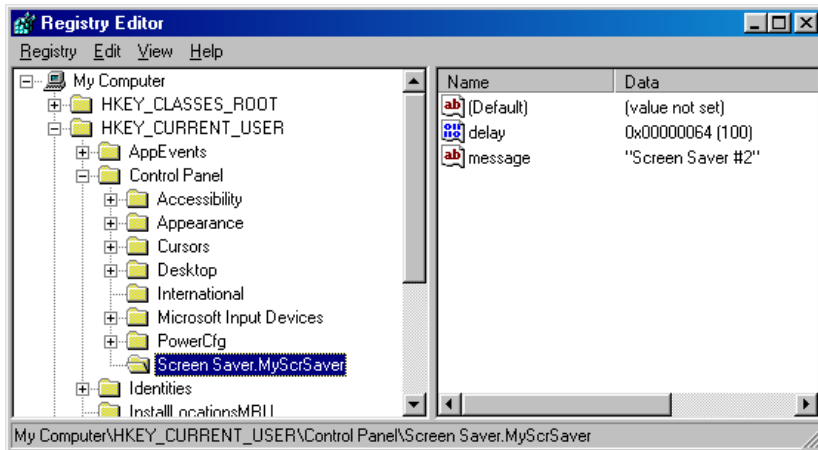



그림 18-2. 화면보호기에 의해 작성된 체계자료기지경로

ScreenSaverProc() 안에서 WM_CREATE통보문을 처리하는 부분을 설명하자. 여기에 서술된 코드는 대화칸에서 체계자료기지를 조작하는 부분과 완전히 동일하다. 물론 사용자에게 체계자료기지의 설정정보를 표시하거나 변경시키게 하는것이 아니라 화면보호기의 동작을 조종하기 위해 설정정보를 사용한다.

여기서 주의해야 할것은 사용자가 설정대화칸을 표시시키지 않고 화면보호기를 실행하는 경우도 있으므로 WM_CREATE통보문을 처리하는 코드에서도 열쇠의 작성과 값의 설정을 할수 있게 되여 있다는것이다. 다시말하면 항상 체계자료기지에 Control Panel\Screen Saver.MyScrSaver라는 열쇠가 존재하며 delay와 message의 값이 있다고 가정하여서는 안된다는것이다.

자체로 해보기

이 장의 목적은 조작체계의 자료기지와 화면보호기의 틀거리를 설명하는것이였다. 더 좋은 화면보호기를 작성하는것은 독자들에게 맡긴다. 화면전체크기의 비트매프를 작성하고 그것을 배경으로 사용하는것으로부터 시작하는것이 쉬울것이다. 물론 배경을 때때로 변화시키는것이 필요하다. 그렇지 않으면 화면보호기로서의 목적을 상실하는것으로 된다. 화상스캐너를 가지고 있다면 수자사진을 표시하여 간단하고 효과적인 화면보호기를 작성할수 있다.

이 장에서 작성한 설정가능한 화면보호기는 실례프로그람이므로 체계자료기지의

HKEY_LOCAL_MACHINE 열쇠의 밑에 프로그람의 기입사항을 작성할수 없었다. 보통 규모가 큰 응용프로그람을 설치할 때는 설치프로그람에 의해 *HKEY_LOCAL_MACHINE* 의 아래에 프로그람의 기입사항이 작성된다.

만일 화면보호기와 그의 아이콘을 설치하는 자체의 설치프로그람을 작성한다면 *HKEY_LOCAL_MACHINE* 의 아래에 기입사항을 추가하는 처리를 진행하여야 한다. *HKEY_LOCAL_MACHINE*의 아래에 응용프로그람의 정보를 추가한다면 다음과 같은 경로를 사용한다.

HKEY_LOCAL_MACHINE \ Software \ YourName \ AppName \ Version

관심을 돌려야 할 또하나의 체계자료기지열쇠는 *HKEY_PERFORMANCE_DATA* 이다. 이 열쇠는 두가지 점에서 다른 내장열쇠들과 다르다. 그것은 체계자료기지용의 함수를 사용하여 호출하여도 체계자료기지에 정보가 보관되지 않는다는것이다. 또한 이 열쇠를 호출하여 현재 체계안의 쏘프트웨어의 성능정보를 얻을수 있다는것이다. 이 열쇠는 응용프로그람을 최적화하는 경우에 활용할수 있다.

제 19 장

차림표의 확장

이 장에서는 Windows 프로그람의 가장 기본적인 요소의 하나인 차림표에 대한 설명으로 다시 돌아 간다. 제 4 장에서는 차림표의 기초적인 내용을 설명하였다. 이 장에서는 차림표를 확장하는 방법에 대해 설명한다.

Windows 의 차림표체계는 세련되고 고급한 기능들을 수많이 제공해준다. 이 장에서는 차림표를 확장하기 위한 두가지 문제를 취급한다. 그것은 동적차림표와 *유동차림표*이다. 이 차림표들은 응용프로그람의 차림표의 모양과 조작성을 확장한다. 또한 *WM_MENURBUTTONUP*통보문을 처리하는 방법에 대해서도 설명한다.

동적차림표

간단한 Windows 응용프로그람에서는 자원파일안에 정적인 차림표를 정의하는 경우가 일반적이지만 보다 고급한 응용프로그람에서는 프로그람의 실행시의 상황에 맞게 동적으로 차림표의 항목을 추가하거나 삭제해야 할 필요가 제기되는 경우도 있다.

실례로 문서편집기에서는 편집중의 파일의 상태에 따라 [File]차림표의 표시항목을 변경하는 경우가 있다.

프로그람의 실행시의 상황에 따라 상태가 변경되는 차림표를 *동적차림표*라고 한다. 동적차림표를 사용하여 프로그람의 현재 상황에 맞게 사용자에게 적절한 차림표항목을 표시할수 있다.

Windows 2000은 프로그람의 실행시에 차림표의 내용을 변경하기 위한 몇가지 API 함수를 제공한다. 이 장의 실례프로그람에서 사용되는 API 함수들은 *InsertMenuItem( )*, *EnableMenuItem( )*, *DeleteMenu( )*, *GetMenu( )* 및 *GetSubMenu( )*함수들이다. 실례프로그람을 작성하기에 앞서 이 함수들부터 설명하기로 하자.

차림표에 항목을 추가

실행시에 차림표의 항목을 추가하려면 *InsertMenuItem( )*을 리용한다. 선언은 다음과 같다.

```
BOOL InsertMenuItem(HMENU hMenu, UINT Where,
                    BOOL How, LPMENUITEMINFO MenuInfo);
```

InsertMenuItem( )에서는 hMenu에 차림표의 손잡이를 설정하여 차림표항목을 추가한다. 새 차림표항목은 Where로 지정한 항목의 앞에 삽입된다. Where의 설정방법이 How에 표시된다.

How가 령이 아닌 경우에는 Where에 새 항목을 삽입하는 위치를 가리키는 색인을 설정한다.(색인은 령으로부터 시작한다.) How가 령인 경우에는 Where에 새 항목을 삽입할 위치를 가리키는 기존항목의 ID를 설정한다.

삽입되는 차림표항목은 MenuInfo가 가리키는 *MENUITEMINFO*구조체에 설정한다. *MENUITEMINFO*구조체의 정의는 다음과 같다.

```
typedef struct tagMENUITEMINFO
{
    UINT cbSize;
    UINT fMask;
    UINT fType;
```

```
        UINT fState;
        UINT wID;
        HMENU hSubMenu;
        HBITMAP hbmpChecked;
        HBITMAP hbmpUnchecked;
        DWORD dwItemData;
        LPSTR dwTypeData;
        UINT cch;
        HBITMAP hbmpItem; // Windows 2000 및 Windows 98 이후의 경우에만
    } MENUITEMINFO;
```

cbSize 에는 MENUITEMINFO 구조체의 크기를 설정한다. fMask 의 값은 차림표의 정보를 설정할 때 MENUITEMINFO 구조체의 어느 성원에 유효한 값이 보관되여 있는가를 가리키게 되여 있다. 이것은 능동으로 할 성원을 결정한다는것이다.(차림표의 정보를 얻을 때는 어느 성원에 유효한 정보가 들어 있는가를 가리킨다.) fMask 는 다음의 몇개 값의 조합으로 된다.

fMask 의 값	유효한 성원
MIIM_BITMAP	hbmpItem(Windows 2000 및 Windows 98 이후의 판에서만 리용)
MIIM_CHECKMARKS	hbmpChecked 및 hbmpUnchecked
MIIM_DATA	dwItemData
MIIM_FTYPE	fType(Windows 2000 및 Windows 98 이후의 판에서만 리용)
MIIM_ID	wID
MIIM_STATE	fState
MIIM_STRING	dwTypeData(Windows
MIIM_SUBMENU	hSubMenu
MIIM_TYPE	fType 및 dwTypeData(Windows 2000 에서는 MIIM_BITMAP, MIIM_FTYPE, MIIM_STRING 을 사용한다.

차림표항목의 종류는 fType 에서 설정한다. 이것은 다음의 몇개 값들의 조합으로 된다.

fType 의 값	의 미
MFT_BITMAP	dwTypeData 의 아래단어에 비트매프의 손잡이를 설정한다. 차림표항목에 비트매프가 표시된다. Windows 2000 에서는 fMask 에 MIIM_BITMAP 를 설정한다.
MFT_MENUBARBREAK	차림표띠의 경우에는 항목을 새로운 행에 표시. 튀여나오기차림표인 경우에는 항목을 다른 렬에 표시하고 띠를 사용하여 분할한다.
MFT_MENUBREAK	분할띠가 사용되지 않는다.는것을 제외하고는 MFT_MENUBARBREAK 와 같다.
MFT_OWNERDRAW	소유자그리기의 항목
MFT_RADIOCHECK	일반적인 차림표검사표식이 아니라 단일선택단추로 항목을 선택한다. hbmpChecked 에 NULL 을 설정하여야 한다.
MFT_RIGHTJUSTIFY	항목을 오른쪽맞추기한다. 다른 항목도 오른쪽맞추기로 된다.(차림표띠의 경우에만)
MFT_RIGHTORDER	차림표가 오른쪽에서부터 왼쪽으로 표시된다. 오른쪽에서 왼쪽으로 쓰는 언어를 지원하기 위한 설정이다.
MFT_SEPARATOR	차림표항목사이에 가로구분선을 긋는다. dwTypeData 와 cch 의 값은 무시된다. 이 종류는 성원에는 설정되지 않는다.
MFT_STRING	dwTypeData 에 차림표항목을 가리키는 문자렬의 지시자를 설정한다. Windows 2000 에서는 fMask 에 MIIM_STRING 을 설정한다.

fState 에는 차림표항목의 상태를 설정한다. 이것은 다음의 값들의 조합으로 된다.

fState 의 값	의 미
MFS_CHECKED	항목이 검사된 상태
MFS_DEFAULT	항목이 체계설정으로 설정된 상태
MFS_DISABLED	항목이 무효한 상태
MFS_ENABLED	항목이 유효한 상태(항목은 체계설정으로 유효한 상태로 된다.)
MFS_GRAYED	항목이 무효한 상태에서 회색으로 표시된다.
MFS_HILITE	항목이 강조표시된 상태

MFS_UNCHECKED	항목이 검사되지 않은 상태
MFS_UNHILITE	항목이 강조표시되지 않은 상태(항목은 체계설정으로 밝게 표시되지 않은 상태로 된다.)

wID 에는 차림표항목의 ID 를 설정한다.

튀여나오기부분차림표를 삽입하는 경우에는 그의 손잡이를 hSubMenu 에 설정한다. 그렇지 않은 경우에는 hSubMenu 에 NULL 을 설정한다.

차림표항목의 검사상태 및 미검사상태를 표시하는 비트매프를 hbmpChecked 및 hbmpUnchecked 에 설정할수 있다. 체계설정의 검사표식을 사용하는 경우에는 이 두개의 성원에 NULL 을 설정한다.

dwItemData 에는 응용프로그람자체의 자료를 설정할수 있다. 이 성원을 사용하지 않은 경우는 령을 설정한다.

차림표에 정보를 설정할 때는 dwTypeData 에 차림표항목을 가리키는 문자렬의 지시자를 설정한다. 차림표정보를 얻을 때는 문자렬을 보관하기 위한 문자배렬에 대한 지시자를 설정한다. 어느 경우에나 dwTypeData 를 사용한다는것을 알려 주기 위해 fMask 에 *MIIM_STRING* 을 설정하여야 한다.

차림표항목을 얻을 때 fMask 에 MIIM_STRING 이 설정되여 있는 경우는 cch 에 문자렬의 길이가 보관된다. 차림표항목을 설정할 때는 cch 의 값이 무시된다.

차림표본문의 왼쪽에 표시될 비트매프를 지정하는 경우는 hbmpItem 에 비트매프의 손잡이를 설정한다. fMask 에는 *MIIM_BITMAP* 를 설정하여야 한다. 이 추가선택항목은 Windows 2000 에 새롭게 추가된것이다.

InsertMenuItem( )은 호출이 성공하면 령이 아닌 값을 돌려 주고 실패하면 령을 돌려 준다.

Windows 2000의 새로운 기능 : Windows 2000 에서는 차림표의 본문과 함께 비트매프를 설정할수 있다.

이식과 관련한 요점 : InsertMenuItem( )은 Windows 3.1 에서 지원되지 않는다. Windows 3.1 에서는 차림표를 동적으로 삽입하기 위해 AppendMenu( ) 또는 InsertMenu( )를 사용한다. 이 함수들은 Windows 2000 에서도 지원하지만 InsertMenuItem( )를 리용할것을 권고한다.

차림표항목의 삭제

차림표항목을 삭제하려면 *DeleteMenu( )*함수를 사용한다. 선언은 다음과 같다.

```
BOOL DeleteMenu(HMENU hMenu, UINT ItemID, UINT How);
```

hMenu 에는 조작대상으로 되는 차림표항목을 설정한다. 삭제할 항목을 ItemID 에 설정한다. How 의 값은 ItemID 의 설정방법을 가리키게 된다. How 가 *MF_BYPOSITION* 인 경우 ItemID 에 삭제할 항목의 색인을 설정한다.

이 색인은 차림표에서 항목의 위치를 나타내는것이므로 선두의 차림표항목이 령으로 된다. How 가 *MF_BYCOMMAND* 인 경우 ItemID 에 삭제할 항목의 ID 를 설정한다. DeleteMenu( )는 호출이 성공하면 령이 아닌 값을 돌려 주며 실패하면 령을 돌려 준다.

삭제된 차림표항목이 튀여나오기부분차림표의 항목인 경우는 튀여나오기부분차림표의 파괴도 진행된다. *DestroyMenu( )*를 호출할 필요는 없다.(*DestroyMenu( )*에 대해서는 제 4 장에서 설명한다.)

차림표의 손잡이를 얻기

차림표항목을 추가하거나 삭제하려면 차림표의 손잡이가 필요하게 된다. 차림표의 손잡이를 얻으려면 *GetMenu( )*함수를 사용한다. 선언은 다음과 같다.

```
HMENU GetMenu(HWND hwnd);
```

GetMenu( )는 hwnd 로 지정된 창문의 차림표손잡이를 돌려 준다. 호출이 실패한 경우에 NULL 을 돌려 준다.

창문의 차림표손잡이를 얻으면 *GetSubMenu( )*함수를 리용하여 부분차림표(튀여나오기차림표)의 손잡이를 얻을수 있다. 선언은 다음과 같다.

```
HMENU GetSubMenu(HMENU hMenu, int ItemPos);
```

hMenu 에는 어미차림표(기본차림표)의 손잡이를 설정한다. ItemPos 에는 목적하는 튀여나오기차림표의 어미창문의 위치를 설정한다.(선두의 위치가 령으로 된다.) 이 함수는 호출이 성공하면 지정된 튀여나오기차림표의 손잡이를 돌려 주며 실패하면 NULL 을 돌려 준다.

차림표의 항목수를 얻기

동적으로 차림표를 작성할 때 차림표안의 항목수를 알고 싶은 경우가 있을것이다. 차림표의 항목수를 얻자면 *GetMenuItemCount( )*함수를 사용한다. 선언은 다음과 같다.

```
int GetMenuItemCount(HMENU hMenu);
```

hMenu 에는 목적하는 차림표의 손잡이를 설정한다. 이 함수는 호출이 성공하면 차림표의 항목수를 돌려 주며 실패하면 -1 을 돌려 준다.

차림표항목을 유효 혹은 무효로 하기

특정한 상황에서만 유효한 상태로 하려는 차림표항목도 있을것이다. 이러한 경우에는 일시적으로 차림표항목을 무효로 하고 후에 그것을 유효로 전환시킨다. 이것을 실현하자면 *EnableMenuItem( )*함수를 사용한다. 선언은 다음과 같다.

```
BOOL EnableMenuItem(HMENU hMenu, UINT ItemID, UINT How);
```

hMenu 에는 차림표의 손잡이를 설정한다. 유효 혹은 무효로 할 차림표항목을 ItemID 에 설정한다. How 의 값은 두가지 용도로 쓰인다.

첫번째 용도는 ItemID 의 설정방법을 지적하는것이다. How 가 *MF_BYPOSITION* 의 경우는 Item_ID 에 차림표항목의 색인을 설정한다. 색인은 차림표에서 항목의 위치를 가리키는것이므로 선두의 차림표항목이 령으로 된다. How 가 *MF_BYCOMMAND* 인 경우는 ItemID 에 차림표항목의 ID 를 설정한다. DeleteMenu( )는 호출이 성공하면 령이 아닌 값을 돌려 주며 실패하면 령을 돌려 준다.

두번째 용도는 차림표항목을 유효 혹은 무효의 어느 상태로 할것인가를 설정하는것이다. 이것은 다음의 값으로 설정된다.

값	의　　미
MF_DISABLED	차림표항목을 무효로 한다.
MF_ENABLED	차림표항목을 유효로 한다.
MF_GRAYED	차림표항목을 무효로 하고 회색표시

How 의 값을 설정하기 위해서는 두가지 용도의 기발을 OR 연산자로 조합한다.

EnableMenuItem( )은 호출이 성공하면 항목의 직전 상태를 돌려 주며 실패하면 -1 을 돌려 준다.

다시 한보 전진

GetMenuItemInfo( )와 SetMenuItemInfo( )

차림표의 상세한 정보를 얻거나 설정하려는 경우도 있을것이다. 그것을 위한 가장 간단한 방법은 *GetMenuItemInfo( )*및 *SetMenuItemInfo( )*를 리용하는것이다. 선언은 다음과 같다.

```
BOOL GetMenuItemInfo(HMENU hMenu, UINT ItemID,
```

```
                  BOOL How, LPMENUITEMINFO MenuInfo);

BOOL SetMenuItemInfo(HMENU hMenu, UINT ItemID,
                  BOOL How, LPMENUITEMINFO MenuInfo);
```

이 함수들은 차림표항목의 상세한 정보를 얻거나 설정하기 위한것이다. 대상으로 되는 차림표항목을 가진 차림표의 손잡이를 hMenu에 설정한다.

차림표항목을 ItemID에 설정한다. ItemID의 설정방법은 How에서 지시된다. How가 령이 아닌 경우는 ItemID에 항목의 색인을 설정한다. How가 령인 경우에는 ItemID에 항목의 ID를 설정한다.

GetMenuItemInfo( )에서는 MenuInfo가 가리키는 MENUITEMINFO구조체에 항목의 현재 정보가 보관된다. SetMenuItemInfo( )에서는 MenuInfo 가 가리키는 구조체에 차림표항목의 정보를 설정한다.

어느 함수나 호출이 성공하면 령이 아닌 값을 돌려 주며 실패하면 령을 돌려 준다. 상상할수 있는것처럼 SetMenuItemInfo( )를 사용하면 차림표항목을 유효 혹은 무효로 할수도 있다. 그러나 이러한 목적으로 SetMenuItemInfo( )나 GetMenuItemInfo( )을 사용하는것은 효률적이지 못하다. 차림표에 대한 구체적인 정보를 관리할 때만 이 함수들을 사용해야 합니다.

차림표항목의 동적추가

기본적인 차림표관리함수들에 대한 설명이 끝났으므로 이 함수들을 실제로 사용해보자. 우선 차림표항목을 동적으로 삽입하거나 삭제해 보자. 여기서는 기본창문에 여러가지 GDI객체를 그리는 간단한 프로그람을 리용한다.

이 프로그람에는 두개의 차림표가 있다. [Options]차림표와 [Draw]차림표이다. [Options]차림표는 사용자가 여러가지 추가선택항목을 선택하도록 하기 위한것이다. [Draw]차림표는 그리려는 객체를 사용자가 선택하도록 하기 위한것이다.

실례 19-1에 [Options]차림표의 항목을 동적으로 추가하거나 삭제하는 실례프로그람을 보여 주었다. WindowFunc( )의 IDM_ADDITEM 및 IDM_DELITEM 의 case 문에 특히 주목하여야 한다. 이 부분에서 차림표항목의 추가와 삭제가 진행된다. 프로그람의 실행결과를 그림 19-1에 주었다.

실례 19-1. Menu 프로그람

```
// 차림표의 동적인 조작
```

```
// 이 정의가 필요한 번역프로그람도 있다.
#define WINVER 0x0500

#include <windows.h>
#include <cstring>
#include <cstdio>
#include "menu.h"

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);

char szWinName[] = "MyWin"; // 창문클라스의 이름

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;
  HACCEL hAccel;

  // 창문클라스를 정의한다.
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst;     // 실체의 손잡이
  wcl.lpszClassName = szWinName; // 창문클라스의 이름
  wcl.lpfnWndProc = WindowFunc;  // 창문함수
  wcl.style = 0;                 // 체계설정의 형식

  wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
  wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
  wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

  wcl.lpszMenuName = "DynMenu"; // 기본차림표

  wcl.cbClsExtra = 0; // 보조기억기령역은 필요 없다.
  wcl.cbWndExtra = 0;
```

```
// 창문의 배경색을 흰색으로 한다.
wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

// 창문클라스를 등록한다.
if(!RegisterClassEx(&wcl)) return 0;

/* 창문클라스가 등록되였으므로
   창문을 작성할수 있다. */
hwnd = CreateWindow(
  szWinName, // 창문클라스의 이름
  "Using Dynamic Memus", // 제목
  WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
  CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 너비는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 높이는 Windows가 결정하게 한다.
  NULL,          // 어미창문은 없다.
  NULL,          // 클라스차림표의 덧쓰기는 하지 않는다.
  hThisInst,     // 실체의 손잡이
  NULL           // 추가파라메터는 없다.
);

// 건반가속기를 적재한다.
hAccel = LoadAccelerators(hThisInst, "DynMenu");

// 창문을 표시한다.
ShowWindow(hwnd, nWinMode);
UpdateWindow(hwnd);

// 통보문순환고리를 작성한다.
while(GetMessage(&msg, NULL, 0, 0))
{
  if(!TranslateAccelerator(hwnd, hAccel, &msg)) {
    TranslateMessage(&msg); // 건반통보를 변환한다.
    DispatchMessage(&msg); // Windows 2000에 조종을 넘긴다.
  }
}
```

```
  return msg.wParam;
}

/* 이 함수는 Windows 2000으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받아 들인다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  HDC hdc;
  RECT rect;
  HMENU hmenu, hsubmenu;
  int response;
  int count;
  MENUITEMINFO miInfo;

  switch(message) {
    case WM_COMMAND:
      switch(LOWORD(wParam)) {
        case IDM_ADDITEM: // 동적으로 차림표항목을 추가한다.
          // 기본차림표의 손잡이를 얻는다.
          hmenu = GetMenu(hwnd);

          // 첫번째 튀여나오기차림표의 손잡이를 얻는다.
          hsubmenu = GetSubMenu(hmenu, 0);

          // 튀여나오기차림표의 항목수를 얻는다.
          count = GetMenuItemCount(hsubmenu);

          // 구분선을 추가한다.
          miInfo.cbSize = sizeof(MENUITEMINFO);
          miInfo.fMask = MIIM_FTYPE;
          miInfo.fType = MFT_SEPARATOR;
          miInfo.fState = 0;
          miInfo.wID = 0;
          miInfo.hSubMenu = NULL;
          miInfo.hbmpChecked = NULL;
          miInfo.hbmpUnchecked = NULL;
```

```
        miInfo.dwItemData = 0;
        miInfo.dwTypeData = 0;
        InsertMenuItem(hsubmenu, count, 1, &miInfo);

        // 새로운 차림표항목을 추가한다.
        miInfo.fMask = MIIM_STRING | MIIM_ID;
        miInfo.wID = IDM_NEW;
        miInfo.dwTypeData = "E&rase (This is New Item)";
        InsertMenuItem(hsubmenu, count+1, 1, &miInfo);

        // 「Add Item」항목을 무효로 한다.
        EnableMenuItem(hsubmenu, IDM_ADDITEM,
                       MF_BYCOMMAND | MF_GRAYED);

        // 「Delete Item」항목을 유효로 한다.
        EnableMenuItem(hsubmenu, IDM_DELITEM,
                       MF_BYCOMMAND | MF_ENABLED);
        break;
      case IDM_DELITEM: // 동적으로 차림표항목을 삭제한다.
        // 기본차림표의 손잡이를 얻는다.
        hmenu = GetMenu(hwnd);

        // 첫번째 튀여나오기차림표의 손잡이를 얻는다.
        hsubmenu = GetSubMenu(hmenu, 0);

        // 새 차림표항목과 구분선을 삭제한다.
        count = GetMenuItemCount(hsubmenu);
        DeleteMenu(hsubmenu, count-1, MF_BYPOSITION | MF_GRAYED);
        DeleteMenu(hsubmenu, count-2, MF_BYPOSITION | MF_GRAYED);

        // 「Add Item」을 유효로 한다.
        EnableMenuItem(hsubmenu, IDM_ADDITEM,
                       MF_BYCOMMAND | MF_ENABLED);

        // 「Delete Item」항목을 무효로 한다.
        EnableMenuItem(hsubmenu, IDM_DELITEM,
                       MF_BYCOMMAND | MF_GRAYED);
        break;
```

```
      case IDM_EXIT:
        response = MessageBox(hwnd, "Quit the Program?",
                              "Exit", MB_YESNO);
        if(response == IDYES) PostQuitMessage(0);
        break;
      case IDM_NEW: // 창문을 소거한다.
        hdc = GetDC(hwnd);
        GetClientRect(hwnd, &rect);
        SelectObject(hdc, GetStockObject(WHITE_BRUSH));
        PatBlt(hdc, 0, 0, rect.right, rect.bottom, PATCOPY);
        ReleaseDC(hwnd, hdc);
        break;
      case IDM_LINES:
        hdc = GetDC(hwnd);
        MoveToEx(hdc, 10, 10, NULL);
        LineTo(hdc, 100, 100);
        LineTo(hdc, 100, 50);
        LineTo(hdc, 50, 180);
        ReleaseDC(hwnd, hdc);
        break;
      case IDM_ELLIPSES:
        hdc = GetDC(hwnd);
        Ellipse(hdc, 100, 100, 300, 200);
        Ellipse(hdc, 200, 100, 300, 200);
        ReleaseDC(hwnd, hdc);
        break;
      case IDM_RECTANGLES:
        hdc = GetDC(hwnd);
        Rectangle(hdc, 100, 100, 24, 260);
        Rectangle(hdc, 110, 120, 124, 170);
        ReleaseDC(hwnd, hdc);
        break;
      case IDM_HELP:
        MessageBox(hwnd, "Try Adding a Menu Item",
                   "Help", MB_OK);
        break;
    }
    break;
```

```
    case WM_DESTROY: // 프로그람을 끝낸다.
      PostQuitMessage(0);
      break;
    default:
      /* 이 switch 문에서 지정된것 이외의 통보문은
         Windows 2000 에 처리를 맡긴다. */
      return DefWindowProc(hwnd, message, wParam, lParam);
  }
  return 0;
}
```

이 프로그람은 아래의 자원파일을 필요로 한다.

```
// 동적차림표
#include <windows.h>
#include "menu.h"

DynMenu MENU
{
  POPUP "&Options"
  {
    MENUITEM "&Add Item\tF2", IDM_ADDITEM
    MENUITEM "&Delete Item\tF3", IDM_DELITEM, GRAYED
    MENUITEM "E&xit\tCtrl+X", IDM_EXIT
  }
  POPUP "&Draw"
  {
    MENUITEM "&Lines\tF4", IDM_LINES
    MENUITEM "&Ellipses\tF5", IDM_ELLIPSES
    MENUITEM "&Rectangles\tF6", IDM_RECTANGLES
  }
  MENUITEM "&Help", IDM_HELP
}

// 차림표의 가속기를 정의
DynMenu ACCELERATORS
{
  VK_F2, IDM_ADDITEM, VIRTKEY
```

```
    VK_F3, IDM_DELITEM, VIRTKEY
    VK_F4, IDM_LINES, VIRTKEY
    VK_F5, IDM_ELLIPSES, VIRTKEY
    VK_F6, IDM_RECTANGLES, VIRTKEY
    VK_F1, IDM_HELP, VIRTKEY
    "^X", IDM_EXIT
}
```

머리부파일 MENU.H 의 내용은 다음과 같다. 여기에는 이 장의 뒤부분에서 작성하는 두 프로그람에서 사용되는 값들도 포함되여 있다.

```
#define IDM_EXIT                100
#define IDM_LINES               101
#define IDM_ELLIPSES            102
#define IDM_RECTANGLES          103
#define IDM_HELP                104

#define IDM_ADDITEM             200
#define IDM_DELITEM             201

#define IDM_NEW                 300
#define IDM_NEW2                301
#define IDM_NEW3                302
```

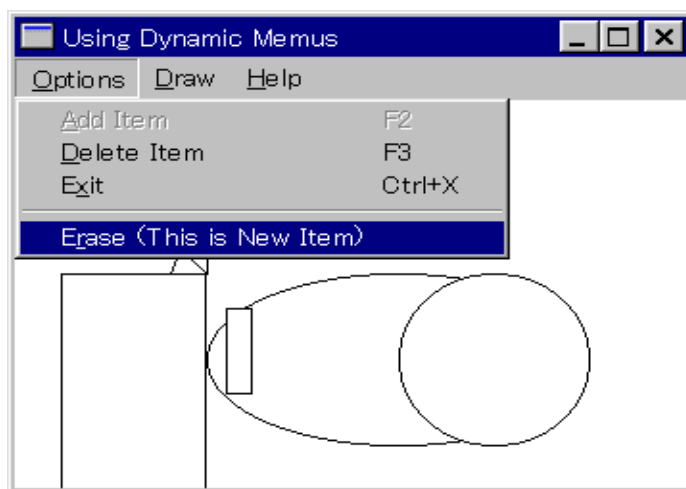



그림 19-1. 차림표항목의 동적추가

동적차림표에 대한 첫 실례프로그람의 상세

프로그람의 대체적인 부분들은 간단하므로 쉽게 리해할수 있다. 프로그람이 기동될 때 [Options]차림표에는 [Add Item], [Delete Item] 및 [Exit]라는 세개의 항목이 있다. 초기상태에서는 [Delete Item]이 회색으로 표시되여 있으므로 그것을 선택할수 없다.

[Erase]라는 차림표항목을 추가하기 위해서 [Add Item]을 선택한다. 차림표에 [Erase]가 동적으로 추가되면 [Delete Item]이 유효로 되며 [Add Item]이 무효로 된다. [Delete Item]을 선택하면 차림표로부터 [Erase]가 삭제되고 [Add Item]이 다시 유효로 되며 [Delete Item]이 본래의 무효상태로 돌아 간다. 이 처리순서에 의해 새로운 차림표항목이 중복되여 추가되거나 삭제되는것을 방지한다.

IDM_ADDITEM 부분을 주의 깊게 살펴 보자. [Options]라는 튀여나오기차림표의 손잡이를 얻는 방법에 주목해야 한다. 우선 GetMenu( )를 리용하여 어미차림표의 손잡이를 얻어야 한다. 여기서는 프로그람의 기본차림표가 어미차림표로 된다.

다음 GetSubMenu( )를 사용하여 첫번째 튀여나오기차림표의 손잡이를 얻는다. 이것은 [Options]이다. 다음 차림표의 항목을 얻는다.

이 실례프로그람에서는 차림표의 구조가 알려 져 있으므로 이 처리순서가 반드시 필요한것은 아니다. 그러나 실제 프로그람에서는 차림표의 구조가 알려 져 있지 않은 경우도 있으므로 이러한 순서를 보여 주고 있다. 마지막으로 차림표에 구분선과 [Erase]항목을 추가하고 있다.

프로그람의 선두에서 WINVER 라는 마크로에 0x0500 이라는 값을 정의하고 있는데 주의해야 한다. 이 마크로에 의해 Windows 2000 에 새로 추가된 머리부파일의 정보가 인용되게 된다. 이 정의를 하지 않으면 Windows 2000 혹은 Windows 98 에서만 유효한 MIIM_STRING 과 같은 옹근수가 인용되지 않는 번역프로그람도 있다.

동적튀여나오기차림표의 작성

이미 있는 차림표에 새로운 항목을 추가하는것만이 아니라 튀여나오기차림표전체를 동적으로 추가할수도 있다.(이것은 실행시에 튀여나오기차림표를 작성한다는것이다.) 먼저 차림표를 작성하고 다음 그것을 이미 있는 차림표에 추가한다. 동적튀여나오기차림표를 작성하려면 *CreatePopupMenu( )*라는 API 함수를 리용한다. 선언은 다음과 같다.

```
HMENU CreatePopupMenu( );
```

이 함수는 비여 있는 차림표를 작성하고 그의 손잡이를 돌려 준다. InsertMenuItem( )을 리용하여 이 차림표에 항목을 추가한다. 차림표의 모든 항목이 추가되였으면 다시 InsertMenuItem( )을 사용하여 그것을 이미 있는 차림표에 삽입한다.

CreatePopupMenu( )를 리용하여 작성된 차림표는 후에 파괴해야 한다. 그러나 차림표가 창문에 관련되여 있는 경우에는 자동적으로 파괴된다. 차림표의 어미차림표가 DeleteMenu( )로 삭제될 때도 자동적으로 파괴된다. *DestroyMenu( )*를 사용하면 동적차림표를 임의로 동적으로 파괴할수 있다.

실례 19-2 의 프로그람은 앞의 실례프로그람을 개조한것이다. 여기서는 [Erase], [Black Pen] 및 [Red Pen] 이라는 세개의 항목을 가진 튀여나오기차림표를 동적으로 작성한다. [Erase]를 선택하면 창문의 내용이 삭제된다. [Black Pen]을 선택하면 검은색의 펜이 선택된다.(이것은 체계설정의 펜이다.) [Red Pen]을 선택하면 붉은색의 펜이 선택된다. 선택된 펜은 [Draw]차림표에서 도형을 그릴 때 사용된다.

이 프로그람에서는 튀여나오기차림표의 작성과 그것을 [Options]차림표에 추가하는 방법에 주목하여야 한다.

실례 19-2. Menu2 프로그람

```
// 튀여나오기차림표의 추가

// 이 정의가 필요한 번역프로그람도 있다.
#define WINVER 0x0500

#include <windows.h>
#include <cstring>
#include <cstdio>
#include "menu.h"

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);

char szWinName[] = "MyWin"; // 창문클라스의 이름

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;
  HACCEL hAccel;
```

```
// 창문클라스를 정의한다.
wcl.cbSize = sizeof(WNDCLASSEX);

wcl.hInstance = hThisInst;      // 실체의 손잡이
wcl.lpszClassName = szWinName; // 창문클라스의 이름
wcl.lpfnWndProc = WindowFunc;  // 창문함수
wcl.style = 0;                 // 체계설정의 형식

wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

wcl.lpszMenuName = "DynPopUpMenu"; // 기본차림표

wcl.cbClsExtra = 0; // 보조기억기령역은 필요 없다.
wcl.cbWndExtra = 0;

// 창문의 배경색을 흰색으로 한다.
wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

// 창문클라스를 등록한다.
if(!RegisterClassEx(&wcl)) return 0;

/* 창문클라스가 등록되였으므로
   창문을 작성할수 있다. */
hwnd = CreateWindow(
  szWinName, // 창문클라스의 이름
  "Adding a Popup Menu", // 제목
  WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
  CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 너비는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 높이는 Windows가 결정하게 한다.
  NULL,          // 어미창문은 없다.
  NULL,          // 클라스차림표의 덧쓰기를 하지 않는다.
  hThisInst,     // 실체의 손잡이
  NULL           // 추가파라메터는 없다.
```

```
  );
  // 건반가속기를 적재한다.
  hAccel = LoadAccelerators(hThisInst, "DynPopUpMenu");

  // 창문을 표시한다.
  ShowWindow(hwnd, nWinMode);
  UpdateWindow(hwnd);

  // 통보문순환고리를 작성한다.
  while(GetMessage(&msg, NULL, 0, 0))
  {
    if(!TranslateAccelerator(hwnd, hAccel, &msg)) {
      TranslateMessage(&msg); // 건반통보를 변환한다.
      DispatchMessage(&msg); // Windows 2000 에 조종을 넘긴다.
    }
  }

  return msg.wParam;
}

/* 이 함수는 Windows 2000 으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받아 들인다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  HDC hdc;
  HMENU hmenu, hsubmenu;
  RECT rect;
  static HMENU hpopup;
  int response;
  int count;
  MENUITEMINFO miInfo;
  static HPEN hCurrentPen, hRedPen;

  switch(message) {
    case WM_CREATE:
      // 붉은색의 펜을 작성한다.
      hRedPen = CreatePen(PS_SOLID, 1, RGB(255, 0, 0));
      // 검은색의 펜을 얻는다.
```

```
        hCurrentPen = (HPEN) GetStockObject(BLACK_PEN);
        break;
    case WM_COMMAND:
        switch(LOWORD(wParam)) {
          case IDM_ADDITEM: // 동적으로 튀여나오기차림표를 추가한다.
            // 기본차림표의 손잡이를 얻는다.
            hmenu = GetMenu(hwnd);

            // 첫번째 튀여나오기차림표의 손잡이를 얻는다.
            hsubmenu = GetSubMenu(hmenu, 0);

            // 튀여나오기차림표의 항목수를 얻는다.
            count = GetMenuItemCount(hsubmenu);

            // 새로운 튀여나오기차림표를 작성한다.
            hpopup = CreatePopupMenu( );

            // 동적튀여나오기차림표에 항목을 추가한다.
            miInfo.cbSize = sizeof(MENUITEMINFO);
            miInfo.fMask = MIIM_STRING | MIIM_ID;
            miInfo.wID = IDM_NEW;
            miInfo.hSubMenu = NULL;
            miInfo.hbmpChecked = NULL;
            miInfo.hbmpUnchecked = NULL;
            miInfo.dwItemData = 0;
            miInfo.dwTypeData = "&Erase";
            InsertMenuItem(hpopup, 0, 1, &miInfo);

            miInfo.dwTypeData = "&Black Pen";
            miInfo.wID = IDM_NEW2;
            InsertMenuItem(hpopup, 1, 1, &miInfo);

            miInfo.dwTypeData = "&Red Pen";
            miInfo.wID = IDM_NEW3;
            InsertMenuItem(hpopup, 2, 1, &miInfo);

            // 구분선을 추가한다.
            miInfo.cbSize = sizeof(MENUITEMINFO);
            miInfo.fMask = MIIM_FTYPE;
            miInfo.fType = MFT_SEPARATOR;
            miInfo.fState = 0;
```

```
        miInfo.wID = 0;
        miInfo.hSubMenu = NULL;
        miInfo.hbmpChecked = NULL;
        miInfo.hbmpUnchecked = NULL;
        miInfo.dwItemData = 0;
        InsertMenuItem(hsubmenu, count, 1, &miInfo);

        // 기본차림표에 튀여나오기차림표를 추가한다.
        miInfo.fMask = MIIM_STRING | MIIM_SUBMENU;
        miInfo.hSubMenu = hpopup;
        miInfo.dwTypeData = "&This is New Popup";
        InsertMenuItem(hsubmenu, count+1, 1, &miInfo);

        // 「Add Popup」항목을 무효로 한다.
        EnableMenuItem(hsubmenu, IDM_ADDITEM,
                    MF_BYCOMMAND | MF_GRAYED);

        // 「Delete Popup」항목을 유효로 한다.
        EnableMenuItem(hsubmenu, IDM_DELITEM,
                    MF_BYCOMMAND | MF_ENABLED);
        break;
    case IDM_DELITEM: // 동적으로 튀여나오기차림표를 삭제한다.
        // 기본차림표의 손잡이를 얻는다.
        hmenu = GetMenu(hwnd);

        // 첫번째 튀여나오기차림표의 손잡이를 얻는다.
        hsubmenu = GetSubMenu(hmenu, 0);

        // 새로운 튀여나오기차림표와 구분선을 삭제한다.
        count = GetMenuItemCount(hsubmenu);
        DeleteMenu(hsubmenu, count-1, MF_BYPOSITION | MF_GRAYED);
        DeleteMenu(hsubmenu, count-2, MF_BYPOSITION | MF_GRAYED);

        // 「Add Popup」을 유효로 한다.
        EnableMenuItem(hsubmenu, IDM_ADDITEM,
                    MF_BYCOMMAND | MF_ENABLED);

        // 「Delete Popup」을 무효로 한다.
        EnableMenuItem(hsubmenu, IDM_DELITEM,
                    MF_BYCOMMAND | MF_GRAYED);
        break;
```

```
        case IDM_EXIT:
          response = MessageBox(hwnd, "Quit the Program?",
                                  "Exit", MB_YESNO);
          if(response == IDYES) PostQuitMessage(0);
          break;
        case IDM_NEW: // 창문을 소거한다.
          hdc = GetDC(hwnd);
          GetClientRect(hwnd, &rect);
          SelectObject(hdc, GetStockObject(WHITE_BRUSH));
          PatBlt(hdc, 0, 0, rect.right, rect.bottom, PATCOPY);
          ReleaseDC(hwnd, hdc);
          break;
        case IDM_NEW2: // 검은색 펜을 선택한다.
          hCurrentPen = (HPEN) GetStockObject(BLACK_PEN);
          break;
        case IDM_NEW3: // 붉은색 펜을 선택한다.
          hCurrentPen = hRedPen;
          break;
        case IDM_LINES:
          hdc = GetDC(hwnd);
          SelectObject(hdc, hCurrentPen);
          MoveToEx(hdc, 10, 10, NULL);
          LineTo(hdc, 100, 100);
          LineTo(hdc, 100, 50);
          LineTo(hdc, 50, 180);
          ReleaseDC(hwnd, hdc);
          break;
        case IDM_ELLIPSES:
          hdc = GetDC(hwnd);
          SelectObject(hdc, hCurrentPen);
          Ellipse(hdc, 100, 100, 300, 200);
          Ellipse(hdc, 200, 100, 300, 200);
          ReleaseDC(hwnd, hdc);
          break;
        case IDM_RECTANGLES:
          hdc = GetDC(hwnd);
          SelectObject(hdc, hCurrentPen);
          Rectangle(hdc, 100, 100, 24, 260);
```

```
            Rectangle(hdc, 110, 120, 124, 170);
            ReleaseDC(hwnd, hdc);
            break;
          case IDM_HELP:
            MessageBox(hwnd, "Try Adding a Menu", "Help", MB_OK);
            break;
        }
        break;
      case WM_DESTROY: // 프로그람을 끝낸다.
        PostQuitMessage(0);
        break;
      default:
        /* 이 switch 문에서 지정된것 이외의 통보문은
           Windows 2000 에 처리를 맡긴다. */
        return DefWindowProc(hwnd, message, wParam, lParam);
    }
    return 0;
  }
```

이 프로그람은 앞의 실례프로그람과 같은 머리부파일 MENU.H 를 사용한다. 그러나 자원파일은 아래에 준것을 사용한다.

```
// 동적튀여나오기차림표
#include <windows.h>
#include "menu.h"

DynPopUpMenu MENU
{
  POPUP "&Options"
  {
    MENUITEM "&Add Popup\tF2", IDM_ADDITEM
    MENUITEM "&Delete Popup\tF3", IDM_DELITEM, GRAYED
    MENUITEM "E&xit\tCtrl+X", IDM_EXIT
  }
  POPUP "&Draw"
  {
    MENUITEM "&Lines\tF4", IDM_LINES
    MENUITEM "&Ellipses\tF5", IDM_ELLIPSES
    MENUITEM "&Rectangles\tF6", IDM_RECTANGLES
```

```
    }
    MENUITEM "&Help", IDM_HELP
  }

  // 차림표의 가속기를 정의한다.
  DynPopUpMenu ACCELERATORS
  {
    VK_F2, IDM_ADDITEM, VIRTKEY
    VK_F3, IDM_DELITEM, VIRTKEY
    VK_F4, IDM_LINES, VIRTKEY
    VK_F5, IDM_ELLIPSES, VIRTKEY
    VK_F6, IDM_RECTANGLES, VIRTKEY
    VK_F1, IDM_HELP, VIRTKEY
    "^X", IDM_EXIT
  }
```

프로그람의 실행결과는 그림 19-2 에 보여 주었다.

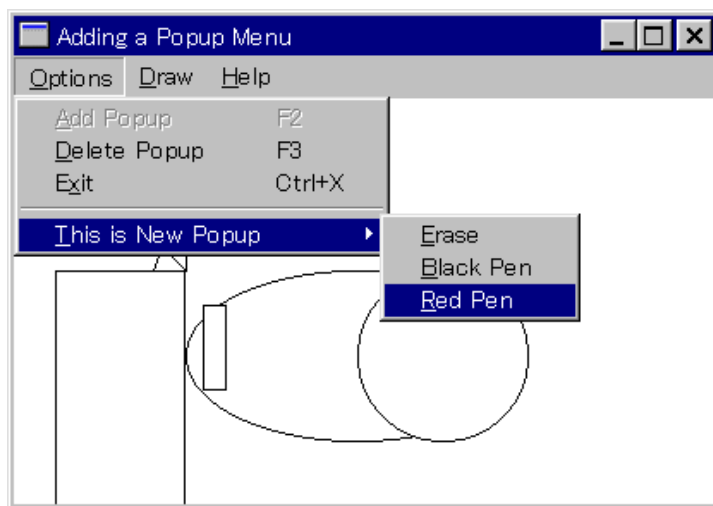



그림 19-2. 차림표항목의 동적추가

프로그람의 다른 부분의 내용은 쉽게 리해할수 있다. 그러나 주의를 돌려야 할 부분도 있다. 그것은 *MIIM_SUBMENU* 기발이 설정되여 있다는것과 튀여나오기차림표를 차림표에 삽입할 때 그의 손잡이를 hSubMenu 에 설정하고 있는것이다.

다시 한보 전진

차림표띠의 변경

차림표에 항목을 추가하는것과 완전히 같은 방법으로 차림표띠에 항목을 추가할수도 있다. 그러나 한가지 처리를 추가하여야 한다. 그것은 변경내용을 표시하기 위해 차림표띠를 다시그리기하는것이다. 그것을 위해 *DrawMenuBar( )*를 리용한다. 선언은 다음과 같다.

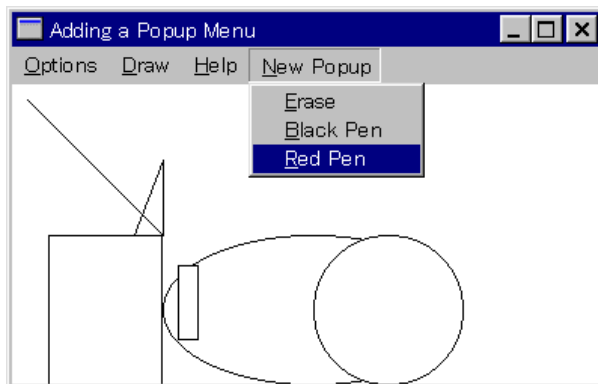
```
BOOL DrawMenuBar(HWND hwnd);
```

hwnd 에 차림표띠가 소속된 창문의 손잡이를 설정한다. 이 함수는 호출이 성공하면 령이 아닌 값을 돌려 주며 실패하면 령을 돌려 준다.

차림표띠내용을 변경하는 시험을 목적으로 튀여나오기차림표의 실례프로그람에서 IDM_ADDITEM 부분을 다음의 프로그람코드로 치환한다.

```
// 차림표띠에 동적으로 튀여나오기차림표를 추가한다.
case IDM_ADDITEM:
  // 차림표띠의 손잡이를 얻는다.
  hmenu = GetMenu(hwnd);

  // 차림표띠의 항목수를 얻는다.
  count = GetMenuItemCount(hmenu);

  // 새로운 튀여나오기차림표를 작성한다.
  hpopup = CreatePopupMenu( );
  // 동적튀여나오기차림표에 항목을 추가한다.
  miInfo.cbSize = sizeof(MENUITEMINFO);
  miInfo.fMask = MIIM_STRING | MIIM_ID;
  miInfo.wID = IDM_NEW;
  miInfo.hSubMenu = NULL;
  miInfo.hbmpChecked = NULL;
  miInfo.hbmpUnchecked = NULL;
  miInfo.dwItemData = 0;
  miInfo.dwTypeData = "&Erase";
  InsertMenuItem(hpopup, 0, 1, &miInfo);

  miInfo.dwTypeData = "&Black Pen";
  miInfo.wID = IDM_NEW2;
  InsertMenuItem(hpopup, 1, 1, &miInfo);
```

```
miInfo.dwTypeData = "&Red Pen";
miInfo.wID = IDM_NEW3;
InsertMenuItem(hpopup, 2, 1, &miInfo);

// 차림표띠에 튀여나오기차림표를 추가한다.
miInfo.fMask = MIIM_STRING | MIIM_SUBMENU;
miInfo.hSubMenu = hpopup;
miInfo.dwTypeData = "&New Popup";
InsertMenuItem(hmenu, count+1, 1, &miInfo);

// 첫번째 튀여나오기차림표의 손잡이를 얻는다.
hsubmenu = GetSubMenu(hmenu, 0);
// 「Add Popup」항목을 무효로 한다.
EnableMenuItem(hsubmenu, IDM_ADDITEM, MF_BYCOMMAND |
                MF_GRAYED);
// 「Delete Popup」항목을 유효로 한다.
EnableMenuItem(hsubmenu, IDM_DELITEM, MF_BYCOMMAND |
                MF_ENABLED);

// 차림표띠를 다시그리기한다.
DrawMenuBar(hwnd);
break;
```

이러한 변경을 진행하면 새로운 튀여나오기차림표가 내리펼침차림표가 아니라 차림표띠에 추가된다. 같은 순서로 IDM_DELITEM 부분에서 차림표띠로부터 튀여나오기차림표를 삭제하는 프로그람코드도 작성해 볼수 있다. 새로운 튀여나오기차림표가 추가된 차림표띠는 다음 그림과 같다.

차림표띠의 내용을 변경시키려는 경우에는 개별적으로 변경을 진행할 대신 *SetMenu( )*를 사용할 수도 있다. SetMenu( )는 창문의 기본차림표전체를 통채로 바꾸는 함수이다. 아래에 선언을 보여 주었다.

```
BOOL SetMenu(HWND hwnd, HMENU hmenu);
```

hwnd에는 대상으로 되는 창문의 손잡이를 설정한다. hmenu에는 새로운 차림표의 손잡이를 설정한다. 낡은 차림표는 *DestroyMenu( )*를 호출하여 파괴한다. 응용프로그람에서 여러개의 기본차림표를 사용할 필요가 있으면 그것들을 미리 작성해 두고 필요에 따라 SetMenu( )를 사용하여 차림표를 반전절환하는것이 합리적이다. 이 방법은 일반적으로 차림표띠를 일일이 변경하는것보다 효률적이다.

기본차림표자체를 동적으로 작성하려는 경우는(이것은 차림표띠인것이다.) *CreateMenu( )*함수를 사용한다. 선언은 다음과 같다.

```
HMENU CreateMenu( );
```

이 함수는 빈 차림표의 손잡이를 돌려 주므로 동적튀여나오기차림표와 같은 방법으로 차림표항목을 추가하여야 한다.

유동차림표의 사용방법

고립형(Stand alone)의 *유동차림표는* 한때 Windows 프로그람작성자들속에서만 알려 져 있었으나 현재는 일반사용자들에게도 충분히 침투하여 그 중요성이 증대되고 있다. 이러한 리유의 하나로서 현재 모든 판본의 Windows 에서는 탁상면에서 마우스의 오른쪽 단추를 찰칵하면 유동차림표가 표시되도록 되여 있다는 사실을 들수 있다.

거의 모든 본격적인 응용프로그람들에서는 유동차림표가 활용되고 있다. 유동차림표기능을 지원하지 않고서는 최신의 Windows 2000 프로그람을 작성할수 없다고 말해도 과언이 아니다.

유동차림표를 상황차림표나 지름차림표라고도 부른다. 이 책에서는 가장 일반적인 호칭인 유동차림표라는 표현을 사용하기로 한다.

유동차림표의 표시

유동차림표를 표시하려면 *TrackPopupMenuEx( )*함수를 사용한다. 선언은 다음과

같다.

```
BOOL TrackPopupMenuEx(HMENU hMenu, UINT Flags, int X, int Y,
                      HWND hwnd, LPTMPARMS OffLimits);
```

hMenu에는 표시하려는 차림표의 손잡이를 설정한다.

Flags에는 여러가지 추가선택항목을 설정할수 있다. 이 파라메터에는 표 19-1에 보여 준 값을 조합하여 설정한다.(상반되는 의미를 가진 기발들은 조합할수 없다.) Flags에 령을 설정할수도 있다. 령인 경우에는 체계설정이 리용된다.

표 19-1. TrackPopupMenuEx( )의 Flags파라메터의 값

Flags 의 값	의 미
TPM-BOTTOMALIGN	Y를 하단으로 하여 유동차림표를 표시
TPM_CENTERALIGN	X를 좌우의 중심으로 하여 유동차림표를 표시
TPM_HORIZONTAL	X, Y로 지정된 위치에 차림표전체를 표시할수 없는 경우는 수평방향의 위치설정을 우선시한다.
TPM_HORNEGANIMATION	오른쪽에서 왼쪽으로의 동화효과를 사용(Windows 2000 및 Windows 98에서만)
TPM_HORPOSANIMATION	왼쪽에서 오른쪽으로의 동화효과를 사용(Windows 2000 및 Windows 98에서만)
TPM_LEFTALIGN	X를 왼쪽 끝으로 하여 유동차림표를 표시(이것은 체계설정이다.)
TPM_LEFTBUTTON	마우스의 왼쪽 단추로 차림표항목을 선택(이것은 체계설정이다.)
TPM_NOANIMATION	동화효과를 사용하지 않음((Windows 2000 및 Windows 98에서만)
TPM_NONOTIFY	차림표는 통지문을 보내지 않는다.
TPM_RETURNCMD	선택된 차림표의 ID가 돌려 진다.
TPM_RIGHTALIGN	X를 오른쪽 끝으로 유동차림표를 표시한다.
TPM_RIGHTBUTTON	마우스의 오른쪽 단추로 차림표항목을 선택한다.
TPM_TOPALIGN	Y를 상단으로 하여 유동차림표를 표시(이것은 체계설정이다.)
TPM_VCENTRALIGN	Y를 아래우의 중심으로하여 유동차림표를 표시
TPM_VERNEGANIMATION	아래에서부터 우에로의 동화효과를 사용(Windows 2000 및 Windows 98에서만)

TPM_VERPOSANIMATION	우에서부터 아래로의 동화효과를 사용(Windows 2000 및 Windows 98 에서만)
TPM_VERTICAL	X, Y 로 지정된 위치에 차림표전체를 표시할수 없는 경우 수직방향의 위치설정을 우선시한다.

화면상에 차림표를 표시시키기 위한 위치를 X 와 Y 에 설정한다. 이 자리표는 화면단위로 되며 창문이나 대화칸의 단위가 아니다. 화면자리표와 창문자리표를 서로 변환하려면 ClientToScreen( )함수나 ScreenToClient( )함수를 리용한다. 체계설정에서는 *TrackPopupMenuEx( )*의 X, Y 를 왼쪽웃모서리의 자리표로 하여 유동차림표가 표시된다. 그러나 Flags 파라메터를 설정하여 이 위치를 변경할수도 있다.

TrackPopupMenuEx( )를 호출하는 창문의 손잡이를 hwnd 에 설정한다.

유동차림표가 화면에 표시되는 령역을 제한할수도 있다. 그러자면 OffLimits 가 가리키는 TPMPARAMS 구조체에 령역범위를 설정한다. TPMPARAMS 구조체의 정의는 다음과 같다.

```
typedef struct tagTPMPARAMS
{
    UINT cbSize;
    RECT reExclude;
}   TPMPARAMS;
```

cbSize 에는 TPMPARAMS 구조체의 크기를 설정한다. rcExclude 에는 제외시키는 령역의 크기를 설정한다. rcExclude 에 설정하는 자리표는 화면단위로 한다. 화면에 표시를 금지하는 령역이 없는 경우에는 OffLimits 에 NULL 을 설정한다.

TrackPopupMenuEx( )는 호출이 성공하면 령이 아닌 값을 돌려 주며 실패하면 령을 돌려 준다. 그러나 Flags 에 TPM_RETURNCMD 가 설정되여 있는 경우에는 선택된 차림표항목의 ID 가 돌려 진다. 이 경우에는 차림표항목이 선택되지 않았을 때 령이 돌려 진다.

이식과 관련한 요점 : Windows 3.1 에서는 유동차림표를 표시하는데 TrackPopupMenu( )함수가 사용되였다. 낡은 프로그람코드를 Windows 2000 에 이식하는 경우에는 이것을 TrackPopupMenuEx( )로 치환하여야 한다.

유동차림표의 실례

실례 19-3 에 준 프로그람은 앞의 프로그람을 개조하여 [Draw]차림표를 *유동차림표*

로 한것이다. 그러므로 [Draw]차림표는 차림표띠에 표시되지 않고 마우스의 오른쪽 단추를 누르면 표시된다. 유동차림표는 마우스의 단추를 눌렀을 때의 마우스 지시자의 위치에 표시된다. 프로그람의 실행결과를 그림 19-3에 보여 주었다.

실례 19-3. Menu3 프로그람

```
// 유동차림표

// 이 정의가 필요한 번역프로그람도 있다.
#define WINVER 0x0500
#define _WIN32_WINNT 0x0500

#include <windows.h>
#include <cstring>
#include <cstdio>
#include "menu.h"

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);

char szWinName[] = "MyWin"; // 창문클라스의 이름

HINSTANCE hInst;

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;
  HACCEL hAccel;

  // 창문클라스를 정의한다.
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst;      // 실체의 손잡이
  wcl.lpszClassName = szWinName; // 창문클라스의 이름
  wcl.lpfnWndProc = WindowFunc;  // 창문함수
  wcl.style = 0;                 // 체계설정의 형식
```

```
wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

wcl.lpszMenuName = "FloatMenu"; // 기본차림표

wcl.cbClsExtra = 0; // 보조기억기령역은 필요 없다.
wcl.cbWndExtra = 0;

// 창문의 배경색을 흰색으로 한다.
wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

// 창문클라스를 등록한다.
if(!RegisterClassEx(&wcl)) return 0;

/* 창문클라스가 등록되였으므로
   창문을 작성할수 있다. */
hwnd = CreateWindow(
  szWinName, // 창문클라스의 이름
  "Using a Floating Popup Menu", // 제목
  WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
  CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 너비는 Windows가 결정하게 한다.
  CW_USEDEFAULT, // 높이는 Windows가 결정하게 한다.
  NULL,          // 어미창문은 없다.
  NULL,          // 클라스차림표의 덧쓰기는 하지 않는다.
  hThisInst,     // 실체의 손잡이
  NULL           // 추가파라메터는 없다.
);

hInst = hThisInst; // 실체의 손잡이를 보관한다.

// 건반가속기를 적재한다.
hAccel = LoadAccelerators(hThisInst, "FloatMenu");

// 창문을 표시한다.
```

```
    ShowWindow(hwnd, nWinMode);
    UpdateWindow(hwnd);

    // 통보문순환고리를 작성한다.
    while(GetMessage(&msg, NULL, 0, 0))
    {
      if(!TranslateAccelerator(hwnd, hAccel, &msg)) {
        TranslateMessage(&msg); // 건반통보를 변환한다.
        DispatchMessage(&msg); // Windows 2000에 조종을 넘긴다.
      }
    }

    return msg.wParam;
  }

  /* 이 함수는 Windows 2000으로부터 호출되여
     통보문대기렬에서 꺼낸 통보문을 받아 들인다.
  */
  LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                              WPARAM wParam, LPARAM lParam)
  {
    HDC hdc;
    HMENU hmenu, hsubmenu;
    RECT rect;
    static HMENU hpopup;
    int response;
    int count;
    MENUITEMINFO miInfo;
    POINT pt;
    static HPEN hCurrentPen, hRedPen;

    switch(message) {
      case WM_CREATE:
        // 붉은색 펜을 작성한다.
        hRedPen = CreatePen(PS_SOLID, 1, RGB(255, 0, 0));
        // 검은색 펜을 얻는다.
        hCurrentPen = (HPEN) GetStockObject(BLACK_PEN);
        break;
```

```
    case WM_COMMAND:
      switch(LOWORD(wParam)) {
        case IDM_ADDITEM: // 동적으로 튀여나오기차림표를 추가한다.
          // 기본차림표의 손잡이를 얻는다.
          hmenu = GetMenu(hwnd);

          // 첫번째 튀여나오기차림표의 손잡이를 얻는다.
          hsubmenu = GetSubMenu(hmenu, 0);

          // 튀여나오기차림표의 항목수를 얻는다.
          count = GetMenuItemCount(hsubmenu);

          // 새로운 튀여나오기차림표를 작성한다.
          hpopup = CreatePopupMenu( );

          // 동적튀여나오기차림표에 항목을 추가한다.
          miInfo.cbSize = sizeof(MENUITEMINFO);
          miInfo.fMask = MIIM_STRING | MIIM_ID;
          miInfo.wID = IDM_NEW;
          miInfo.hSubMenu = NULL;
          miInfo.hbmpChecked = NULL;
          miInfo.hbmpUnchecked = NULL;
          miInfo.dwItemData = 0;
          miInfo.dwTypeData = "&Erase";
          InsertMenuItem(hpopup, 0, 1, &miInfo);

          miInfo.dwTypeData = "&Black Pen";
          miInfo.wID = IDM_NEW2;
          InsertMenuItem(hpopup, 1, 1, &miInfo);

          miInfo.dwTypeData = "&Red Pen";
          miInfo.wID = IDM_NEW3;
          InsertMenuItem(hpopup, 2, 1, &miInfo);

          // 구분선을 추가한다.
          miInfo.cbSize = sizeof(MENUITEMINFO);
          miInfo.fMask = MIIM_FTYPE;
          miInfo.fType = MFT_SEPARATOR;
```

```
        miInfo.fState = 0;
        miInfo.wID = 0;
        miInfo.hSubMenu = NULL;
        miInfo.hbmpChecked = NULL;
        miInfo.hbmpUnchecked = NULL;
        miInfo.dwItemData = 0;
        InsertMenuItem(hsubmenu, count, 1, &miInfo);

        // 기본차림표에 튀여나오기차림표를 추가한다.
        miInfo.fMask = MIIM_STRING | MIIM_SUBMENU;
        miInfo.hSubMenu = hpopup;
        miInfo.dwTypeData = "&This is New Popup";
        InsertMenuItem(hsubmenu, count+1, 1, &miInfo);

        // 「Add Popup」항목을 무효로 한다.
        EnableMenuItem(hsubmenu, IDM_ADDITEM,
                       MF_BYCOMMAND | MF_GRAYED);

        // 「Delete Popup」항목을 유효로 한다.
        EnableMenuItem(hsubmenu, IDM_DELITEM,
                       MF_BYCOMMAND | MF_ENABLED);
        break;
    case IDM_DELITEM: // 동적으로 튀여나오기차림표를 삭제한다.
        // 기본차림표의 손잡이를 얻는다.
        hmenu = GetMenu(hwnd);

        // 첫번째 튀여나오기차림표의 손잡이를 얻는다.
        hsubmenu = GetSubMenu(hmenu, 0);

        // 새로운 튀여나오기차림표와 구분선을 삭제한다.
        count = GetMenuItemCount(hsubmenu);
        DeleteMenu(hsubmenu, count-1, MF_BYPOSITION | MF_GRAYED);
        DeleteMenu(hsubmenu, count-2, MF_BYPOSITION | MF_GRAYED);

        // 「Add Popup」항목을 무효로 한다.
        EnableMenuItem(hsubmenu, IDM_ADDITEM,
                       MF_BYCOMMAND | MF_ENABLED);

        // 「Delete Popup」항목을 유효로 한다.
        EnableMenuItem(hsubmenu, IDM_DELITEM,
```

```
                              MF_BYCOMMAND | MF_GRAYED);
          break;
        case IDM_EXIT:
          response = MessageBox(hwnd, "Quit the Program?",
                                "Exit", MB_YESNO);
          if(response == IDYES) PostQuitMessage(0);
          break;
        case IDM_NEW: // 창문을 소거한다.
          hdc = GetDC(hwnd);
          GetClientRect(hwnd, &rect);
          SelectObject(hdc, GetStockObject(WHITE_BRUSH));
          PatBlt(hdc, 0, 0, rect.right, rect.bottom, PATCOPY);
          ReleaseDC(hwnd, hdc);
          break;
        case IDM_NEW2: // 검은색 펜을 선택한다.
          hCurrentPen = (HPEN) GetStockObject(BLACK_PEN);
          break;
        case IDM_NEW3: // 붉은색 펜을 선택한다.
          hCurrentPen = hRedPen;
          break;
        case IDM_LINES:
          hdc = GetDC(hwnd);
          SelectObject(hdc, hCurrentPen);
          MoveToEx(hdc, 10, 10, NULL);
          LineTo(hdc, 100, 100);
          LineTo(hdc, 100, 50);
          LineTo(hdc, 50, 180);
          ReleaseDC(hwnd, hdc);
          break;
        case IDM_ELLIPSES:
          hdc = GetDC(hwnd);
          SelectObject(hdc, hCurrentPen);
          Ellipse(hdc, 100, 100, 300, 200);
          Ellipse(hdc, 200, 100, 300, 200);
          ReleaseDC(hwnd, hdc);
          break;
        case IDM_RECTANGLES:
          hdc = GetDC(hwnd);
```

```
            SelectObject(hdc, hCurrentPen);
            Rectangle(hdc, 100, 100, 24, 260);
            Rectangle(hdc, 110, 120, 124, 170);
            ReleaseDC(hwnd, hdc);
            break;
          case IDM_HELP:
            MessageBox(hwnd, "Try Pressing Right Mouse Button",
                       "Help", MB_OK);
            break;
        }
        break;
      case WM_RBUTTONDOWN: // 유동차림표를 튀여나오기시킨다.

        // 창문자리표를 화면자리표로 변환한다.
        pt.x = LOWORD(lParam);
        pt.y = HIWORD(lParam);
        ClientToScreen(hwnd, &pt);

        // 「Draw」차림표의 손잡이를 얻는다.
        hmenu = LoadMenu(hInst, "Draw");

        // 첫번째 튀여나오기차림표의 손잡이를 얻는다.
        hsubmenu = GetSubMenu(hmenu, 0);

        // 유동차림표를 표시한다.
        TrackPopupMenuEx(hsubmenu, 0, pt.x, pt.y,
                         hwnd, NULL);
        DestroyMenu(hmenu);
        break;
      case WM_DESTROY: // 프로그람을 끝낸다.
        PostQuitMessage(0);
        break;
      default:
        /* 이 switch 문에서 지정된것 이외의 통보문은
           Windows 2000 에 처리를 맡긴다. */
        return DefWindowProc(hwnd, message, wParam, lParam);
    }
    return 0;
}
```

이 프로그람은 앞의 프로그람과 같은 MENU.H 파일을 사용한다. 자원파일의 내용은 다음과 같다.

```
// 유동차림표
#include <windows.h>
#include "menu.h"

FloatMenu MENU
{
  POPUP "&Options"
  {
    MENUITEM "&Add Popup\tF2", IDM_ADDITEM
    MENUITEM "&Delete Popup\tF3", IDM_DELITEM, GRAYED
    MENUITEM "E&xit\tCtrl+X", IDM_EXIT
  }
  MENUITEM "&Help", IDM_HELP
}

// 이 차림표가 유동차림표로 된다.
Draw MENU
{
  POPUP "&Draw" {
    MENUITEM "&Lines\tF4", IDM_LINES
    MENUITEM "&Ellipses\tF5", IDM_ELLIPSES
    MENUITEM "&Rectangles\tF6", IDM_RECTANGLES
  }
}

// 차림표의 가속기를 정의한다.
FloatMenu ACCELERATORS
{
  VK_F2, IDM_ADDITEM, VIRTKEY
  VK_F3, IDM_DELITEM, VIRTKEY
  VK_F4, IDM_LINES, VIRTKEY
  VK_F5, IDM_ELLIPSES, VIRTKEY
  VK_F6, IDM_RECTANGLES, VIRTKEY
  VK_F1, IDM_HELP, VIRTKEY
  "^X", IDM_EXIT
}
```

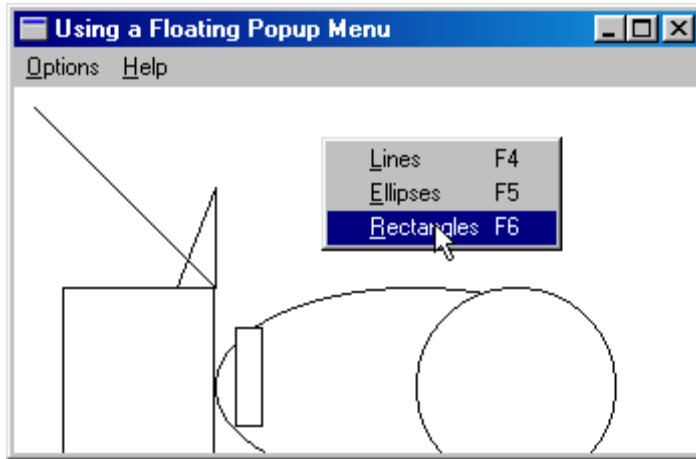



그림 19-3. 유동차림표프로그람의 실행결과

이 프로그람의 자원파일에서는 [Draw]차림표가 기본차림표띠의 항목으로 되여 있지 않다는 사실에 주의를 돌려야 한다. [Draw]차림표는 독립적인 차림표로서 정의되여 있다. 그러므로 차림표가 호출될 때까지 표시되지 않는다.

유동차림표프로그람의 상세

프로그람의 대체적인 부분은 앞의 실례프로그람과 같다. 단지 WM_RBUTTONDOWN 부분의 프로그람코드에 주의를 돌리면 된다. 이 부분에서 [Draw]차림표를 기동하고 있다. 아래에 다시 한번 프로그람코드를 보여 주었다.

```
    case WM_RBUTTONDOWN: // 유동차림표를 튀여나오게 한다.

  // 창문자리표를 화면자리표로 변환한다.
      pt.x = LOWORD(lParam);
      pt.y = HIWORD(lParam);
      ClientToScreen(hwnd, &pt);

      // 「Draw」차림표의 손잡이를 얻는다.
      hmenu = LoadMenu(hInst, "Draw");
      // 첫번째 튀여나오기차림표의 손잡이를 얻는다.
      hsubmenu = GetSubMenu(hmenu, 0);

      // 유동차림표를 표시한다.
      TrackPopupMenuEx(hsubmenu, 0, pt.x, pt.y,
                       hwnd, NULL);
```

```
        DestroyMenu(hmenu);
        break;
```

이 프로그람코드는 마우스의 오른쪽 단추가 눌리웠을 때의 마우스지시자의 위치를 왼쪽 웃모서리의 자리표로 하여 [Draw]차림표를 튀여나오기시킨다. 그런데 *TrackPopupMenuEx( )*에서는 화면단위가 사용되므로 *ClientToScreen( )*을 사용하여 창문단위로 표시된 마우스위치를 화면단위로 변환하여야 한다. LoadMenu( )를 사용하여 차림표를 적재하면 그의 첫번째(이 실례에서는 한개밖에 없다.) 튀여나오기차림표의 손잡이를 돌려 준다. 이러한 처리가 완료되면 차림표를 표시할수 있다.

*_WIN32_WINNT*라는 마크로가 0x0500라는 값으로 정의된 점에도 주의를 돌려야 한다. 이것은 TrackPopupMenu( )에서 사용되는 모든 기발들의 마크로를 포함시키기 위해 필요하게 된다.

튀여나오기차림표의 동화효과

Windows 2000에서는 흥미 있는 기능이 유동차림표에 추가되여 있다. 그것은 차림표가 표시될 때 *동화효과*를 제공할수 있다는것이다.

차림표의 동화효과란 차림표가 표시되는 순차를 설정하는것이다. 우에서부터 아래로, 아래에서부터 우로, 왼쪽에서부터 오른쪽으로 또는 오른쪽에서부터 왼쪽으로 순차를 설정할수 있다. 표 19-1에 제시한 기발들가운데서 동화효과를 지정하는것은 아래의 다섯개이다.

TPM_HORNEGANIMATION　　TPM_HORPOSANIMATION
TPM_VERNEGANIMATION　　TPM_VERPOSANIMATION
TPM_NOANIMATION

TPM_NOANIMATION은 동화효과를 사용하지 않고 순간적으로 차림표를 튀여나오기시킨다.

아래에서부터 우로의 동화효과를 시험해 보기 위하여 앞의 실례프로그람에서 TrackPopupMenuEx( )를 호출하는 부분을 아래와 같이 변경해 본다.

```
    TrackPopupMenuEx(hsubmenu, TPM_VERNEGANIMATION, pt.x,
                         pt.y, hwnd, NULL);
```

이 변경을 진행하면 마우스의 오른쪽 단추를 찰칵하였을 때 튀여나오기차림표가 화면의 밑에서부터 웃쪽을 향해 표시되게 된다. 차림표의 동화효과는 튀여나오기차림표에 특수한 효과를 주는 가장 간단한 방법의 하나이다.

Windows 2000의 새로운 기능 : 차림표의 동화효과는 Windows 2000에 새롭게 추가된 것이다. 이 기능은 Windows NT 4.0에서는 지원되지 않는다.

WM_MENURBUTTONUP 통보문의 처리

이미 설명한바와 같이 최신판의 모든 Windows 에서는 마우스의 오른쪽 단추를 누르면 유동차림표가 표시된다. 또한 도움말창문이 표시되는 응용프로그람도 있다. Windows NT 4.0에서는 이 기능이 하나의 례외 즉 차림표를 제외하고는 모든 사용자대면부의 요소에 적용되고 있다.

Windows 2000에서는 차림표항목에도 적용되였다. Windows 2000에서는 차림표항목우에서 오른쪽 단추를 찰칵하면 *WM_MENURBUTTONUP*통보문이 발생된다. 이 통보문을 처리하면 선택된 항목에 관한 도움말정보를 표시하거나 부분차림표를 튀여나오기시킬수 있다. WM_MENURBUTTONUP 통보문을 처리하는것은 프로그람에 최신의 조작성을 제공하는데 적당한 방법의 하나이다.

WM_MENURBUTTONUP통보문이 생성될 때는 IParam에 차림표의 손잡이가 보관되며 wParam에 오른쪽 찰칵한 항목의 색인이 보관된다. 이러한 정보를 얻는 방법을 보여 주기 위하여 앞에서 보여 준 실례프로그람의 WindowsFunc( )의 switch 문에 아래의 프로그람코드를 추가해 본다. 이 코드는 [Options]차림표의 오른쪽찰칵을 처리하는 코드이다.

```
case WM_MENURBUTTONUP:
    // 「Options」차림표의 오른쪽 찰칵을 처리한다.
    if((HMENU)lParam == GetSubMenu(GetMenu(hwnd), 0))
      switch(wParam) {
        case 0: MessageBox(hwnd, "Add Popup",
                  "Right Click", MB_OK);
          break;
        case 1: MessageBox(hwnd, "Delete Popup",
                  "Right Click", MB_OK);
          break;
        case 2: MessageBox(hwnd, "Exit",
                  "Right Click", MB_OK);
      }
```

```
        break;
```

프로그람코드를 추가하고 나서 [Options]차림표를 내리펼치기하고 [Add Popup]을 마우스의 오른쪽 단추로 찰칵해 본다. 그러면 통보칸이 표시되게 된다.

프로그람코드의 처리내용을 설명해 보자. 차림표항목의 우에서 마우스의 오른쪽 단추를 찰칵하면 WM_MENURBUTTONUP 통보문이 발송되며 그때 lParam에는 차림표의 손잡이가 wParam에는 차림표항목의 색인이 보관되여 있다. lParam에 보관된 값으로부터 [Options]차림표의 손잡이를 얻을수 있으며 wParam 에 보관된 색인으로부터 어느 통보칸을 표시하면 좋겠는가를 판단할수 있다.

물론 실제의 응용프로그람에서는 WM_MENURBUTTONUP 통보문에 대한 응답으로서 부분차림표를 표시하거나 도움말창문을 표시하는것으로 될것이다. 통보칸은 실례를 보여 주기 위해 표시하였을뿐이다.

Windows 2000 의 새로운 기능 : Windows NT 4.0 의 프로그람을 이식할 때는 WM_MENURBUTTONUP 통보문을 처리하는 기능을 추가하면 좋을것이다.

자체로 해보기

우선 간단히 시험해 볼것이 있다. 이 장의 첫 실례프로그람에서는 [Options]차림표의 [Erase]항목이 사용자에 의해 수동으로 추가되거나 삭제되였다. 이 수법은 실례를 보여 주기 위한것일뿐이다.

프로그람에서 자동적으로 [Erase]항목을 추가하거나 삭제하게 할수도 있다. 실례로 프로그람의 기동시와 같은 창문의 내용이 비여 있는 경우에는 [Erase]를 무효로 해둔다. 사용자가 창문에 어떤 그리기를 진행하였다면 [Erase]항목을 유효로 한다. 사용자가 창문의 내용을 없애면 다시 [Erase] 항목을 무효로 한다. 이렇게 [Erase]항목의 상태를 자동적으로 설정하는것은 실제적인 응용프로그람의 동적차림표에서 사용되고 있는것이다.

차림표를 삽입할 때 MFT_RADIOCHECK 형식을 설정하고 그 효과를 확인해 본다.

또한 차림표항목에 비트매프를 추가해 본다. 그러자면 MENUITEMINFO 구조체의 hbmpItem성원에 비트매프의 손잡이를 설정한다. 또한 fMask에는 *MIIM_BITMAP*를 설정해야 한다.

TrackPopupMenuEx( )함수에 설정할수 있는 여러가지 추가선택항목을 시험해 본다. 실례로 표시를 금지하는 령역의 정의를 진행해 본다. 또한 차림표의 배치를 달리 설정해 본다.

그리고 유동차림표와 도움말체계를 통합해 본다. 어떠한 조작기능으로 하면 좋겠는가는 지금 많이 쓰이고 있는 응용프로그람에서 진행되는 처리를 조사해 보면 좋을것이다.

제 20 장

동적련결서고와 보안

제1장에서 시작한 긴 로정을 거쳐 어느덧 마지막 장에 이르렀다. 지금까지의 장들에서 설명한 내용을 모두 리해하였다면 Windows 2000 프로그람을 작성할수 있게 되였다고 말할수 있다. 이 장에서 설명하게 되는 자체의 동적련결서고(DLL)를 작성하는 방법과 Windows 2000 의 보안체계에 대한 지식을 소유하면 Windows 2000 프로그람을 보다 원만하게 작성할수 있다. 이것들을 자세히 설명하는것은 불가능하지만 기초지식을 알아 두는 것은 중요하다.

이 장에서는 매 항목에 대하여 간단히 설명하기로 한다.

동적련결서고의 작성

Win32 API 서고는 동적련결서고로서 제공되고 있다. 이것은 응용프로그람에서 API 함수를 사용하여도 그 함수의 코드는 응용프로그람의 객체파일에 포함되지 않는다는것을 의미한다. 응용프로그람에는 API 함수의 적재명령만이 들어 있다. 실행시에 프로그람이 적재될 때 실제의 API 함수가 련결된다.

사용자들도 이와 같은 방식의 자체의 동적련결서고를 작성하여 리용할수 있다. 뒤에서 보게 되겠지만 DLL 의 작성방법은 결코 어렵지 않다. 우선 정적련결과 동적련결의 차이점에 대한 설명으로부터 시작하기로 하자.

동적련결과 정적련결

Windows 는 *정적련결*과 *동적련결*이라는 두 종류의 련결형식을 지원한다. 정적련결은 번역시에 진행된다. 정적련결되는 함수의 코드는 프로그람의 실행(EXE)파일에 물리적으로 추가된다. 정적련결되는 함수는 .OBJ 또는 *.LIB 파일*안에 들어 있다.

실례로 대규모의 프로그람을 분할번역하여 작성할 때는 .EXE 파일의 작성시에 련결프로그람이 개개 모듈의 .OBJ 파일을 결합한다. 이 경우에는 작성된 .EXE 파일안에 모든 .OBJ 파일안에 있는 코드들이 들어 있다. .LIB 파일로서 제공되는 C/C++의 실행시 서고함수도 정적형식이며 사용자들이 작성한 프로그람에 결합된다.

이와는 달리 동적련결은 번역할 때가 아니라 프로그람의 실행시에 진행되며 동적련결되는 함수의 코드가 프로그람의 .EXE 파일에 추가되지 않는다. 동적련결되는 함수는 사용자가 작성한 프로그람과 독립적으로 존재하는 *.DLL 파일*안에 보관되여 있다. DLL 안에 있는 함수가 프로그람의 실행시에 결합된다. 사용자가 작성한 프로그람에는 DLL 을 적재하기 위한 작은 코드가 련결되여 있지만 함수자체는 련결되지 않는다.

동적련결서고를 작성하는 리유

어째서 자체의 *동적련결서고*를 작성할 필요가 제기되는가고 생각할수도 있다. 소규모의 프로그람에서는 그렇게 할 필요가 없다. 번역할 때 프로그람에 모든 함수를 련결하는 편이 간단하기때문이다. 그러나 기능을 몇개의 *모듈*에 분할하여 작성하는 대규모의 프로그람인 경우에는 큰 가치가 있다. 그 리유는 다음과 같다.

① DLL 에 함수를 보관하면 매개의 모듈에 포함되는 함수가 중복되지 않게 된다. 그러면 매개 모듈의 크기를 작게 할수 있으며 디스크공간을 절약할수 있다.

② DLL 을 사용하면 프로그람의 갱신이 쉽게 된다. 정적련결서고의 경우는 그 안에 보관된 함수의 기능이 변경되면 그 함수를 사용하고 있는 모든 모듈을 다시 련결하여야

한다.

동적련결서고의 경우는 DLL 파일만을 재번역하는것으로 해결된다. DLL 을 사용하는 모든 응용프로그람들은 다음에 실행될 때는 새로운 판의 함수를 사용하게 된다.

③ DLL 을 사용하면 프로그람을 부분적으로 수정하는것이 쉽게 된다. 실례로 프로그람에서 치명적인 오유가 나타난 경우는 응용프로그람전체가 아니라 동적련결서고만을 수정하여 그것을 배포하는것이 효률적이다.

이것은 우주비행선이나 무인감시체계 등과 같은 원격환경에서 운영되는 프로그람에서 특히 유효하게 된다.

물론 불리한 점이 없는것도 아니다. 독자적인 DLL 을 사용하게 되면 프로그람은 두개이상의 모듈에 분할되게 된다. 이렇게 되면 프로그람의 관리가 시끄럽게 되며 어떤 문제를 일으킬 가능성도 있다. 실례로 만약 동적련결서고와 응용프로그람사이에 정합이 되지 않으면 문제점이 발생할것이다. 그러나 대규모의 프로그람을 작성하는 경우에는 동적련결서고의 우점이 그의 부족점을 압도하게 되는것이다.

동적련결서고의 기초

DLL 의 실례프로그람을 작성하기에 앞서 DLL 의 작성방법과 사용방법상의 규칙을 설명해 두자. 첫번째 규칙은 DLL 에 보관되여 있는 프로그람으로부터 호출되는 함수는 *수출*(Export)되여야 한다는것이다. 두번째 규칙은 DLL 에 보관된 함수를 사용하기 위해서는 그것을 *수입*(Import)해야 한다는것이다.

C/C++에서는 dllexport 라는 열쇠단어로 수출을 진행하며 dllimport 라는 열쇠단어로 수입을 진행한다. dllexport 와 dllimport 는 Microsoft Visual C++나 다른 몇가지 번역프로그람에서 지원되는 확장열쇠단어이다.

이식과 관련한 요점 : Windows 3.1 등의 16bit 판본의 Windows 에서는 DLL 에서 수출되는 함수를 .DEF 파일의 EXPORTS 구획에서 정의하여야 했다. 지금도 이 방법을 리용할수 있지만 dllexport 를 사용하는 방법이 간단하다.

dllexport 와 dllimport 의 두개의 열쇠단어는 단독으로 사용할수 없다. *__declspec* 라는 다른 하나의 확장열쇠단어와 함께 사용해야 한다. 아래에 일반적인 구문을 표시한다.

__declspec(specifier)

specifier 에는 보관클라스를 설정한다. DLL 의 경우는 specifier 가 dllexport 또는 dllimport 로 된다. 실례로 MyFunc( )라는 함수를 수출하는 경우에는 다음과 같이 한다.

```
__declspec (dllexport) void MyFunc(int a)
```

수입되거나 수출되는 함수의 선언을 간단히 하기 위해 대다수의 프로그람작성자들은 __declspec 를 사용하는 긴 구문이 아니라 마크로를 사용한다. 실례로 다음과 같은 마크로이다.

```
#define DllExport  __declspec (dllexport)
```

이 마크로를 사용하면 아래와 같은 간단한 구문으로 MyFunc( )를 수출할수 있다.

```
DllExport void MyFunc(int a)
```

C++의 프로그람으로서 번역된 DLL 을 C 언어의 프로그람에서도 사용할수 있게 하려는 경우에는 "C"련결지정을 추가할 필요가 있다. 아래에 구문의 형식을 보여 주었다.

```
#define DllExport  extern "C" __declspec (dllexport)
```

이렇게 하면 C++독자의 이름장식이 진행되는것을 억제할수 있다. 이름장식이란 형정보를 추가하여 함수이름을 내부적으로 변경해 버리는것이다. 이것은 다른 클라스의 성원함수나 다른 이름공간의 함수 등으로 다중정의된 함수를 식별하기 위한 대책안이다.

이 책의 실례프로그람에서는 이 대책안으로 인한 문제를 막기 위해 "C"련결지정을 설정한다.(C 언어 프로그람을 번역하는 경우에는 번역프로그람에 의해 무시되므로 extern "C"를 지정할 필요는 없다.)

DLL을 번역할 때는 번역프로그람에 DLL을 작성하기 위한 설정을 하여야 한다. 그를 위한 가장 간단한 방법은 새로운 프로젝트를 작성할 때 DLL 프로젝트를 선택하는것이다. 번역프로그람의 추가선택을 어떻게 설정하면 좋은가를 알수 없는 경우에는 번역프로그람의 지도서를 참고해야 한다.

DLL 을 번역하면 두 종류의 파일이 작성된다. 하나는 함수를 보관한 DLL 본체로서 파일의 확장자는 .DLL 이다. 다른 하나는 함수의 적재정보를 보관한것이며 파일의 확장자는 .LIB 이다. DLL 을 사용하는 프로그람에는 이 .LIB 파일을 련결하여야 한다.

.DLL 파일은 그것을 사용하는 응용프로그람이 참조할수 있는 등록부에 보관하여야 한다. Windows 2000 은 응용프로그람이 존재하는 등록부, 현재 등록부, 표준적인 DLL 이 보관되여 있는 등록부, 환경변수 PATH 에 설정된 등록부의 순서로 DLL 을 탐색한다.

실험적으로 DLL 을 작성하는 경우에는 표준적인 DLL 들이 보관되여 있는 등록부가 아니라 응용프로그람이 존재하는 등록부에 DLL 을 보관해야 한다. 그렇게 해야 부주의로 체계가 사용하는 DLL 을 변경시키는것을 막을수 있다.

프로그람이 필요한 DLL 을 찾을수 없는 경우에는 기동되지 않으며 오유를 표시하는 통보칸이 표시된다.

간단한 동적련결서고의 실례

아주 간단한 동적련결서고를 작성해 보자. *ShowMouseLoc( )*라는 한개의 함수만을 가지는 DLL 프로그람코드를 실례 20-1 에 보여 주었다. 이 함수는 단추가 눌리웠을 때의 마우스의 위치를 표시하는 기능을 수행한다. 마우스의 단추가 눌리웠을 때의 통보문 lParam 에는 통보문이 생성되였을 때의 마우스의 위치가 보관되여 있다는것을 이미 배웠다. 마우스의 위치를 표시하기 위해서는 이 값과 장치상황을 *ShowMouseLoc( )*의 파라메터로 전달한다. ShowMouseLoc( )를 수출함수로서 선언한다는데 주의해야 한다.

실례 20-1. MyDll 프로그람

```
// 간단한 DLL
#include <windows.h>
#include <cstring>

#define DllExport extern "C" __declspec (dllexport)

/* 이 함수는 마우스의 단추가 눌리웠을 때의
    마우스의 위치를 표시한다.

   hdc:    마우스의 위치를 표시할 장치상황을
           설정한다.
   lParam: 마우스의 단추가 눌리웠을 때의 lParam 의 값
            을 설정한다.
*/
DllExport void ShowMouseLoc(HDC hdc, LPARAM lParam)
{
  char str[80];

  wsprintf(str, "Button is down at %d, %d",
           LOWORD(lParam), HIWORD(lParam));
  TextOut(hdc, LOWORD(lParam), HIWORD(lParam),
          str, strlen(str));
}
```

우선 이 프로그람을 MYDLL.CPP 라는 파일이름으로 작성한다. 다음 DLL 프로젝

트를 작성한다. Visual C++를 사용하는 경우에는 [New]대화칸의 [Project] 표쪽에서 [Win32 Dynamic Link Library]를 선택한다. 프로젝트에 MYDLL.CPP 를 추가하고 서고를 번역하면 MYDLL.DLL 및 MYDLL.LIB 라는 두개의 파일이 생성된다.

이미 설명한것처럼 .DLL 파일은 동적련결서고의 본체이다. .LIB 파일은 이 동적련결서고를 사용하는 응용프로그람에 련결해야 하는 적재정보를 보관한 파일이다.

머리부파일의 작성

프로그람에서 표준적인 서고함수를 사용할 때와 같이 DLL 에 보관된 함수를 사용하는 경우에도 함수의 선언을 하여야 한다. DLL 에 보관된 함수의 선언을 하는 간단한 방법은 DLL 용의 머리부파일을 작성해 두는것이다. 실례로 MYDLL.DLL 을 위한 머리부파일이라면 아래와 같이 한다. 이것을 MYDLL.H 라는 파일이름으로 작성한다.

```
#define DllImport extern "C" __declspec (dllimport)

DllImport void ShowMouseLoc(HDC hdc, LPARAM lParm);
```

동적련결서고의 사용법

DLL 을 번역하여 그것을 적절한 등록부에 보관하고 DLL 용의 머리부파일을 작성하면 DLL 을 사용할 준비는 끝났다. MYDLL.DLL 을 사용하는 간단한 프로그람을 아래에 보여 주었다. 이 프로그람은 마우스의 왼쪽 단추 혹은 오른쪽 단추가 눌리웠을 때의 마우스의 위치를 표시하는 프로그람이다. 프로그람의 실행결과를 그림 20-1 에 준다. 이 프로그람을 번역할 때는 련결목록에 MYDLL.LIB 를 반드시 설정해야 한다. 프로그람을 실행하면 MYDLL.DLL 로부터 *ShowMouseLoc( )*가 자동적으로 적재된다.

```
// DLL 에 보관된 ShowMouseLoc( )의 사용

#include <windows.h>
#include <cstring>
#include "mydll.h"

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);

char szWinName[] = "MyWin"; // 창문클라스의 이름

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
```

```
{
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;

  // 창문클라스를 정의한다.
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst;      // 실체의 손잡이
  wcl.lpszClassName = szWinName; // 창문클라스의 이름
  wcl.lpfnWndProc = WindowFunc;  // 창문함수
  wcl.style = 0;                 // 체계설정의 형식

  wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
  wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
  wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

  wcl.lpszMenuName = NULL; // 클라스차림표는 없다.
  wcl.cbClsExtra = 0;       // 보조기억기령역은 필요 없다.
  wcl.cbWndExtra = 0;

  // 창문의 배경색을 흰색으로 한다.
  wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

  // 창문클라스를 등록한다.
  if(!RegisterClassEx(&wcl)) return 0;

  /* 창문클라스가 등록되였으므로
     창문을 작성할수 있다. */
  hwnd = CreateWindow(
    szWinName, // 창문클라스의 이름
    "Demonstrate a DLL", // 제목
    WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
    CW_USEDEFAULT, // X자리표는 Windows가 결정하게 한다.
    CW_USEDEFAULT, // Y자리표는 Windows가 결정하게 한다.
    CW_USEDEFAULT, // 너비는 Windows가 결정하게 한다.
    CW_USEDEFAULT, // 높이는 Windows가 결정하게 한다.
    NULL,          // 어미창문은 없다.
```

```
    NULL,          // 차림표는 없다.
    hThisInst,     // 실체의 손잡이
    NULL           // 추가파라메터는 없다.
  );

  // 창문을 표시한다.
  ShowWindow(hwnd, nWinMode);
  UpdateWindow(hwnd);

  // 통보문순환고리를 작성한다.
  while(GetMessage(&msg, NULL, 0, 0))
  {
    TranslateMessage(&msg); // 건반통보를 변환한다.
    DispatchMessage(&msg);  // Windows 2000 에 조종을 넘긴다.
  }

  return msg.wParam;
}

/* 이 함수는 Windows 2000 으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받아 들인다.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  HDC hdc;

  switch(message) {
    case WM_RBUTTONDOWN: // 오른쪽 단추를 처리한다.
      hdc = GetDC(hwnd); // 장치상황을 얻는다.
      ShowMouseLoc(hdc, lParam); // DLL 에 보관된 함수를 호출한다.
      ReleaseDC(hwnd, hdc); // 장치상황을 해제한다.
      break;
    case WM_LBUTTONDOWN: // 왼쪽 단추를 처리한다.
      hdc = GetDC(hwnd); // 장치상황을 얻는다.
      ShowMouseLoc(hdc, lParam); // DLL 에 보관된 함수를 처리한다.
      ReleaseDC(hwnd, hdc); // 장치상황을 해제한다.
      break;
```

```
      case WM_DESTROY: // 프로그람을 끝낸다.
        PostQuitMessage(0);
        break;
      default:
        /* 이 switch 문에서 지정된것 이외의 통보문은
           Windows 2000 에 처리를 맡긴다. */
        return DefWindowProc(hwnd, message, wParam, lParam);
    }

    return 0;
  }
```

그림 20-1. 첫 DLL 프로그람의 실행결과

DllMain( )의 사용방법

DLL 에는 어떤 초기화처리나 완료처리가 필요한것들도 있다. 이러한 처리를 실현하기 위하여 DLL 이 초기화되거나 완료될 때 호출되는 *DllMain( )*라는 이름의 함수를 DLL 안에 작성한다. 이 함수는 DLL 의 원천파일안에 써넣는다.

그러나 이 함수를 작성하지 않는 다고 해도 번역프로그람에 의해 암시적처리를 진행하는 코드가 DLL 에 추가된다. 그러므로 앞절에서 작성한 DLL 에서 *DllMain( )*을 서술하지 않아도 문제가 생기지 않았던것이다. 물론 아무리 소규모인 DLL 이라고 해도 모두 내부에 DllMain( )을 가지고 있다. 아래에 선언을 보여 주었다.

```
BOOL WINAPI DllMain(HANDLE hInstance, DWORD What,
```

LPVOID NotUsed);

Windows 2000 이 *DllMain( )*을 호출했을 때 hInstance 에는 DLL 실체의 손잡이가 설정되며 What 에는 발생한 사건의 내용이 설정된다. NotUsed 는 이미 예약된것이다.

What 는 다음의 몇가지 값으로 된다.

값	의 미
DLL_PROCESS_ATTACH	프로세스가 DLL 의 사용을 개시했다.
DLL_PROCESS_DETACH	프로세스가 DLL 을 해제하였다.
DLL_THREAD_ATTACH	프로세스가 새로운 스레드를 작성했다.
DLL_THREAD_DETACH	프로세스가 스레드를 파괴했다.

What 의 값이 *DLL_PROCESS_ATTACH* 인 경우에는 프로세스가 DLL 을 적재하였다는것을 의미한다. 정확히는 DLL 이 프로세스의 주소공간에 넘기기(Mapping)되였다는 것이다. 이 사건에 대한 응답으로서 DllMain( )이 령을 돌려 주면 프로세스는 DLL 의 배속을 중지한다. DLL 을 사용하는 여러개의 프로세스가 있어도 DLL_PROCESS_ATTACH 를 파라메터로 하여 DllMain( )이 호출되는것은 한번만이다.

What 의 값이 *DLL_PROCESS_DETACH* 인 경우는 프로세스가 DLL 을 필요로 하지 않게 되였다는것을 의미한다. 이것은 프로세스자체가 완료될 때 발생하는 사건이다. 이 사건은 DLL 이 해제될 때도 발생한다.

이미 DLL 에 배속되여 있는 프로세스가 새로운 스레드를 작성하면 *DLL_THREAD_ATTACH* 를 파라메터로 하여 DllMain( )이 호출된다. 스레드가 파괴되면 *DLL_THREAD_DETACH* 를 파라메터로 하여 DllMain( )이 호출된다. 여러개의 스레드의 배속 및 분리 통보문이 한 프로세스에서 발생되는 경우도 있다. DLL_PROCESS_ATTACH 통보문을 처리할 때를 제외하고는 DllMain( )의 돌림값은 무시된다.

일반적인 *DllMain( )*에서는 그것이 호출되였을 때의 What 의 값에 따라 결정되는 처리를 진행한다. 결정된 처리라는것은 령이 아닌 값을 돌려 주는것뿐이다.

이식과 관련한 요점 : Win16 용의 DLL 에서는 DllMain( )대신에 LibMain( )이 사용된다. 낡은 프로그람을 이식하는 경우에는 변경이 필요하다.

MYDLL 에 DllMain( )을 추가

DllMain( )이 호출되는 시점을 확인하기 위해 아래에 보여 주는 DllMain( )을

MYDLL.CPP에 추가해 보자.

```
// 간단한 DLL
#include <windows.h>
#include <cstring>

#define DllExport extern "C" __declspec (dllexport)

/* 이 함수는 초기화 및 완료시에
    호출된다.  */
BOOL WINAPI DllMain(HANDLE hInstance, DWORD what,
                    LPVOID Notused)
{
  switch(what) {
    case DLL_PROCESS_ATTACH:
      MessageBox(HWND_DESKTOP, "Process attaching DLL.",
                 "DLL Action", MB_OK);
      break;
    case DLL_PROCESS_DETACH:
      MessageBox(HWND_DESKTOP, "Process detaching DLL.",
                 "DLL Action", MB_OK);
      break;
    case DLL_THREAD_ATTACH:
      MessageBox(HWND_DESKTOP, "Thread attaching DLL.",
                 "DLL Action", MB_OK);
      break;
    case DLL_THREAD_DETACH:
      MessageBox(HWND_DESKTOP, "Thread detaching DLL.",
                 "DLL Action", MB_OK);
      break;
  }
  return 1;
}

/* 이 함수는 마우스의 단추가 눌리웠을 때의
   마우스의 위치를 표시한다.

   hdc:    마우스의 위치를 표시할 장치상황을
```

```
              설정한다.
    lParam: 마우스의 단추가 눌리워 졌을 때의 lParam 의 값을
              설정한다.
*/
DllExport void ShowMouseLoc(HDC hdc, LPARAM lParam)
{
  char str[80];

  wsprintf(str, "Button is down at %d, %d",
           LOWORD(lParam), HIWORD(lParam));
  TextOut(hdc, LOWORD(lParam), HIWORD(lParam),
          str, strlen(str));
}
```

DllMain()이 호출될 때마다 그때의 사건내용이 통보칸에 표시된다. MessageBox()의 첫 파라메터에 *HWND_DESKTOP* 를 설정하는것을 주의해야 한다. HWND_DESKTOP 는 화면을 가리키는것이므로 통보칸을 응용프로그람과 독립시켜 표시할수 있다.

또 하나 보충적으로 알아 두어야 할것은 DllMain()은 응용프로그람으로부터 호출되는것이 아니므로 MYDLL.DLL 로부터 수출되지 않는다. Windows 만이 DllMain( )을 호출한다.

DllMain( )의 실례

실례 20-2 의 프로그람은 DllMain( )의 실례이다. 앞에서 본 프로그람과 마찬가지로 마우스의 왼쪽 단추 또는 오른쪽 단추가 눌리웠을 때 마우스의 위치를 표시하는 ShowMouseLoc( )함수를 호출한다. 그러나 임의의 건을 누르면 스레드가 하나 작성되는 기능이 추가되여 있다.

이 프로그람을 실행하면 DllMain( )이 호출되는가를 확인해 본다. 프로그람의 실행결과를 그림 20-2 에 보여 주었다. 프로그람에 MYDLL.LIB 를 련결하는것을 잊지 말아야 한다.

실례 20-2. DllMain 프로그람

```
// DLL 초기화의 실례


#include <windows.h>
#include <cstring>
#include <cstdio>
```

```
#include "mydll.h"

#define MAX 10000

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);

DWORD WINAPI MyThread(LPVOID param);

char szWinName[] = "MyWin"; // 창문클라스의 이름

char str[255] = ""; // 표시할 문자렬을 보관한다.

DWORD Tid1; // 스레드 ID

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;

  // 창문클라스를 정의한다.
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst;      // 실체의 손잡이
  wcl.lpszClassName = szWinName; // 창문클라스의 이름
  wcl.lpfnWndProc = WindowFunc;  // 창문함수
  wcl.style = 0;                 // 체계설정의 형식

  wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // 큰 아이콘
  wcl.hIconSm = NULL; // 큰 아이콘의 축소판을 사용한다.
  wcl.hCursor = LoadCursor(NULL, IDC_ARROW);  // 유표의 형식

  wcl.lpszMenuName = NULL; // 클라스차림표는 없다.
  wcl.cbClsExtra = 0;       // 보조기억기렁역은 필요 없다.
  wcl.cbWndExtra = 0;

  // 창문의 배경색을 흰색으로 한다.
```

```
  wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

  // 창문클라스를 등록한다.
  if(!RegisterClassEx(&wcl)) return 0;

  /* 창문클라스가 등록되였으므로
     창문을 작성할수 있다. */
  hwnd = CreateWindow(
    szWinName, // 창문클라스의 이름
    "Using DllMain", // 제목
    WS_OVERLAPPEDWINDOW, // 창문의 형식은 표준으로 한다.
    CW_USEDEFAULT, // X 자리표는 Windows 가 결정하게 한다.
    CW_USEDEFAULT, // Y 자리표는 Windows 가 결정하게 한다.
    CW_USEDEFAULT, // 너비는 Windows 가 결정하게 한다.
    CW_USEDEFAULT, // 높이는 Windows 가 결정하게 한다.
    NULL,          // 어미창문은 없다.
    NULL,          // 차림표는 없다.
    hThisInst,     // 실체의 손잡이
    NULL           // 추가파라메터는 없다.
  );

  // 창문을 표시한다.
  ShowWindow(hwnd, nWinMode);
  UpdateWindow(hwnd);

  // 통보문순환고리를 작성한다.
  while(GetMessage(&msg, NULL, 0, 0))
  {
    TranslateMessage(&msg); // 건반통보를 변환한다.
    DispatchMessage(&msg);  // Windows 2000 에 조종을 넘긴다.
  }

  return msg.wParam;
}

/* 이 함수는 Windows 2000 으로부터 호출되여
   통보문대기렬에서 꺼낸 통보문을 받아 들인다.
*/
```

```
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  HDC hdc;
  PAINTSTRUCT ps;

  switch(message) {
    case WM_RBUTTONDOWN: // 오른쪽 단추를 처리한다.
      hdc = GetDC(hwnd); // 장치상황을 얻는다.
      ShowMouseLoc(hdc, lParam); // DLL에 보관된 함수를 처리한다.
      ReleaseDC(hwnd, hdc); // 장치상황을 해제한다.
      break;
    case WM_LBUTTONDOWN: // 왼쪽 단추를 처리한다.
      hdc = GetDC(hwnd); // 장치상황을 얻는다.
      ShowMouseLoc(hdc, lParam); // DLL에 보관된 함수를 호출한다.
      ReleaseDC(hwnd, hdc); // 장치상황을 해제한다.
      break;
    case WM_CHAR: // 건이 눌리웠다면 스레드를 기동한다.
      CreateThread(NULL, 0,
                   (LPTHREAD_START_ROUTINE)MyThread,
                   (LPVOID) hwnd, 0, &Tid1);
      break;
    case WM_PAINT:
      hdc = BeginPaint(hwnd, &ps);
      strcpy(str, "Press a key to start a thread.");
      TextOut(hdc, 1, 1, str, strlen(str));
      EndPaint(hwnd, &ps);
      break;
    case WM_DESTROY: // 프로그람을 끝낸다.
      PostQuitMessage(0);
      break;
    default:
      /* 이 switch문에서 지정된것 이외의 통보문은
         Windows 2000에 처리를 맡긴다. */
      return DefWindowProc(hwnd, message, wParam, lParam);
  }

  return 0;
```

```
}

// 또하나의 스레드
DWORD WINAPI MyThread(LPVOID param)
{
  int i;
  HDC hdc;

  for(i=0; i<MAX; i++) {
    sprintf(str, "In thread: loop # %5d ", i);
    hdc = GetDC((HWND) param);
    TextOut(hdc, 1, 20, str, strlen(str));
    ReleaseDC((HWND) param, hdc);
  }

  return 0;
}
```

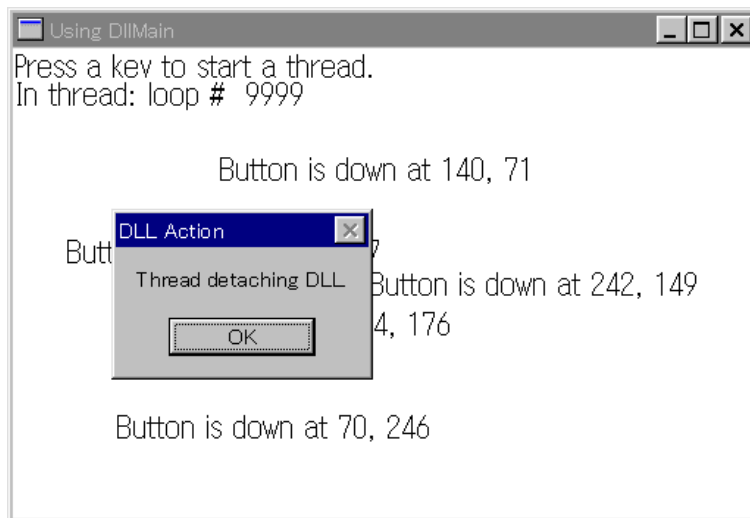



그림 20-2. DllMain( )의 실례

프로그람이 기동할 때에 [Process attaching DLL]라는 통보칸이 표시된다. 임의의 건을 누르면 새로운 스레드가 기동되고 [Thread Attaching DLL]라는 통보칸이 표시된다. 스레드가 완료되면 [Thread Detaching DLL]라는 통보칸이 표시된다. DllMain( )이 언제 무엇때문에 호출되는가를 알아 보기 위해 이 프로그람에서 이것저것 시험해 본다. 독자적인 DLL 이 지금 당장은 필요되지 않을지도 모르지만 사실 여러가지 상황에서 쓸모가 있다.

다시 한보 전진

실행시동적련결(Run-time Dynamic Link)

DLL 에 보관된 함수를 사용할 때는 원천코드안에서 함수이름을 지정하여 호출하는것이 일반적이다. 이것은 이 책의 실례프로그람에서 API 를 호출할 때나 ShowMouseLoc( )를 호출할 때 쓰이는 수법이다. 이 경우에는 함수를 보관한 DLL 이 응용프로그람과 함께 적재된다. 이것은 적재시동적련결(Load-time Dynamic Link)이라고 한다. 그러나 실행시에 DLL 을 적재하는것도 가능하다.

DLL 에 보관된 함수를 명시적으로 호출하지 않은 경우는 프로그람의 실행시에 DLL 이 적재되는것이 아니다. 필요에 따라 DLL 에 보관된 함수를 호출할수도 있다. 그러자면 프로그람에서 DLL 을 적재하고 나서 호출하려는 함수의 지시자를 얻는다. 이것을 실행시동적련결이라고 한다. 실행시동적련결을 하기 위해서는 아래의 세개 함수를 리용한다.

```
HMODULE LoadLibrary(LPCSTR DllName);
BOOL FreeLibrary(HMODULE hMod);
FARPROC GetProcAddress(HMODULE hMod, LPCSTR FuncName);
```

LoadLibrary( )는 DllName 에서 지정된 DLL 을 적재하고 그의 손잡이를 돌려 준다. *FreeLibrary( )*는 불필요하게 된 DLL 을 해제한다. GetProcAddress( )는 hMod 에서 지정된 DLL 에 보관된 함수의 이름을 FuncName 에서 지정하면 그 함수의 지시자를 돌려 준다. 이 지시자를 사용하여 목적하는 함수를 호출할수 있다. 그러나 주의해야 한다. 지정하는 함수는 DLL 에 있는 함수의 이름과 일치해야 한다. C++의 이름장식에 의하여 함수이름이 변경되는 경우도 있다. 이것을 방지하기 위해 "C"련결지정을 사용할수 있다.

실행시동적련결을 시험해 보기 위해서 DLL 의 두번째 실례프로그람에서 아래와 같은 변경을 해 보자. 우선 대역손잡이를 추가한다.

```
HMODULE hLib;
```

다음에 WinMain( )의 통보문순환고리 밑에 다음의 프로그람코드를 삽입한다.

```
hlib = LoadLibrary("MYDLL.DLL");
if (!hlib) MessageBox(hwnd, "Cannot Load Library",
                      "Error", MB_OK);
```

다음 WindowFunc( )안에 함수의 지시자를 추가한다.

```
void (*f) (HDC, LPARAM);
```

마지막으로 마우스통보문의 처리를 다음과 같이 변경한다.

```
case WM_RBUTTONDOWN:     // 오른쪽 단추를 처리
     hdc = GetDC(hwnd);     // 장치상황을 얻는다.

     // ShowMouseLoc( )의 지시자를 얻는다.
          f = (void (*) (HDC, LPARAM)) GetProcAddress(hlib,
               ShowMouseLoc");
     // ShowMouseLoc( )를 호출한다.
     (*f) (hdc, lParam);
     ReleaseDC(hwnd, hdc); // 장치상황을 해제한다.
     break;
case WM_LBUTTONDOWN: // 왼쪽 단추를 처리한다.
     hdc = GetDC(hwnd);      // 장치상황을 얻는다.
// ShowMouseLoc( )의 지시자를 얻는다.
f = (void (*)(HDC, LPARAM)) GetProcAddress(hlib, "ShowMouseLoc");

// ShowMouseLoc( )를 호출한다.
(*f)(hdc, lparam);

ReleaseDC(hwnd, hdc); // 장치상황을 얻는다.
break;
```

WM_DESTROY 통보문을 처리하는 부분에 FreeLibrary( )를 호출하는 프로그람코드를 추가한다. 이렇게 변경한 프로그람은 DLL 을 동적으로 적재하고 ShowMouseLoc( )를 직접적으로가 아니라 지시자를 통해서 호출한다.

보통은 적재시동적련결을 사용해도 무방하지만 실행시동적련결에는 확장기능이 있다. 실례로 현재의 콤퓨터에 목적하는 DLL 이 존재한다면 그 DLL 의 기능을 사용하고 존재하지 않는다면 다른 수단을 사용할수 있다.

보안기능

제 1 장에서 설명한것처럼 Windows 2000 에는 C2 보안준위에 해당하는 *보안기능*이 갖추어 져 있다. Windows 2000 의 보안부분체계에서는 몇가지 보안기능을 제공하고 있다. 그가운데서 기본적인 두가지 기능으로서 콤퓨터본체 및 여러가지 객체에 대한 호출조종이 있다. 전자는 Windows 2000 에로의 등록을 통제하는것이며 후자는 호출권을 통제하

는것이다.

보안설정을 진행하는 객체의 종류에는 파일(NTFS 만 가능), 파이프, 프로세스, 스레드, 체계자료기지열쇠,인쇄기,탁상면 및 동기화객체(신호기 등)가 있다. 이 기능들밖에도 Windows 2000 의 보안체계는 보안사건의 감시, 기억기의 보호 및 조작체계의 보호(변경방지) 등의 기능도 제공하고 있다.

보안기능은 Win32API 서고에 편입되여 있다. 보안기능은 얇은 층과 같은것으로서 응용프로그람에 큰 부담을 주지 않는다. 보안기능을 특별히 필요로 하지 않는 응용프로그람에는 영향을 미치지 않는다.

실례로 지금까지 거슬러 온 장들에서 사용해 온 API 함수들가운데는 파라메터에 보안서술자를 설정할수 있는 함수들이 있었다. 그때는 파라메터에 NULL 을 설정했으며 그렇게 함으로써 체계설정의 보안서술자가 리용되게 되였던것이다.

실천적인 문제로서 많은 Windows 프로그람작성자들에게 있어서 보안기능을 엄밀히 설정할 필요는 없을것이다. 그러나 기본적인 틀거리만은 알아 두어야 할 필요가 있다.

용어의 정의

보안기능을 습득하기 위한 첫 걸음은 몇가지 용어의 의미를 리해하는것이다. 보안서술자는 무엇이 객체를 호출하고 어떻게 객체를 호출할수 있는가를 결정하는것이다. 이것은 *SECURITY_DESCRIPTOR* 구조체에 의해 지적된다.

프로그람에서 이 구조체를 직접 조작하는것은 아니다. 대신 여러가지 API 함수들을 사용하여 구조체의 설정내용을 얻거나 설정을 진행한다.

보안서술자는 소유자나 그의 단체와 관련한 정보를 포함하고 있다. *임의접근조종목록* 또는 체계접근조종목록도 포함되여 있다.

소유자와 단체는 *보안식별자*(SID)에서 지적된다. SID 는 여러가지 보안정보를 가지는 유일한 값이다.

접근조종목록(ACL)은 객체를 호출하고 있는 사용자를 관리하는 목록이다. ACL 은 *접근조종입구*(Access Control Entry)로 구성된다. ACE 는 객체에 대한 사용자 또는 단체의 호출권을 설정하는것이다. ACL 은 호출등록을 감시하는데 사용된다.

ACL 에는 임의접근조종목록(DACL)과 *체계접근조종목록*(SACL)의 두 종류가 있다. DACL 은 객체를 호출하고 있는 사용자 혹은 호출의 종류를 설정하는것이다. 그러므로 객체의 DACL 이 객체에 대한 사용자나 단체의 호출권을 결정하는것으로 된다.

객체에 DACL 이 없는 경우는 모든 사용자에게 모든 호출권이 부여된다. 객체의 소유자는 DACL 의 내용을 변경시킬수 있다.

SACL 은 호출사건의 등록을 감시하는것이다. 체계관리자(System Administrator)권한을 가진 사용자는 SACL 의 내용을 변경시킬수 있다.

접근통표(Access token)는 사용자를 식별하고 사용자의 보안속성을 가리키는것이다. 그가운데는 사용자의 SID, 호출권 및 체계설정의 DACL 이 포함되여 있다.

프로세스에는 호출권의 모임이 정의되여 있다. 호출권은 64bit 옹근수값의 *LUID* 로

표시된다. *LUID*는 *Locally Unique Identifier*의 략칭이다. 호출권의 실례로서는 체계시간의 설정이나 체계의 정지(shut down)를 진행하는 권리 등이 있다. 경우에 따라서는 호출에 특수한 과제를 실행시키기 위한 호출권을 변경하는것도 있다.

보안구조

Windows 2000의 보안부분체계가 객체에 대한 호출을 통제하는 기본적인 구조를 설명해 보자. 보안설정이 가능한 객체는 객체의 소유자 및 객체의 작성자를 정의하는 *보안서술자*와 관련이 지어진다.

사용자가 등록하면 사용자를 식별하기 위한 접근통표가 주어 진다. 이것은 사용자가 소속된 단체를 정의하는것이기도 하다. 사용자에 의해 기동된 프로그람은 접근통표의 복사본을 가진다. 그러므로 사용자가 객체를 호출하기 위해서는 객체의 호출을 허가 받아야 하는것이 아니라 호출의 종류에 따르는 호출권을 얻어야 한다. 이것이 사용자와 사용자가 가지는 호출권이 프로세스에 련결되는 구조이다.

프로세스가 객체를 호출하려고 하면 사용자의 접근통표가 객체의 DACL 과 비교된다. 목록안의 ACE 가 접근통표와 일치하면 호출이 허가된다. 그렇지 않은 경우는 호출이 거부된다.

등록가입(logon)의 호출은 그자체가 *보안회계관리자*(SAM-Security Account Manager)에 의해 관리된다. 등록가입에는 사용자명과 통과암호의 입력이 필요하. 이 통과암호는 체계전체에 대한 기본적인 보안기능을 제공하는것으로 되여 있다.

보안과 관련된 API

Windows 2000 에는 보안기능을 조작하는 API 함수들이 대단히 많다. 다음에 그중 주되는 함수들만을 보여 주었다.

함　　수	기　　능
AdjustTokenPrivileges( )	이미 있던 호출권을 유효 혹은 무효로 한다.
GetAce( )	ACE 의 지시자를 얻는다.
GetFileSecurity( )	파일의 보안설정을 얻는다.
GetSecurityDescriptorDacl( )	보안서술자의 DACL 의 지시자를 얻는다.
GetSecurityDescriptorSacl( )	보안서술자의 SACL 의 지시자를 얻는다.
GetSecurityInfo( )	객체의 손잡이를 주고 보안서술자의 복사를 얻는다.
InitializeSecurityDescriptor( )	보안서술자를 초기화한다.
LogonUser( )	등록가입을 진행한다.
SetFileSecurity( )	파일의 보안을 설정한다.
SetSecurityDescriptorDacl( )	보안서술자에 DACL 을 추가한다.
SetSecurityDescriptorSacl( )	보안서술자에 SACL 을 추가한다.

이 책을 끝내며

Windows 2000 은 지금까지 개발된 쏘프트웨어체계들중에서 가장 거대하고 복잡한 체계의 하나이다. Windows 2000 의 모든 내용을 다 설명할수는 없다. 이 책에서는 망, 자료압축, 동화상 등의 항목은 취급할수 없었다.

이외에도 례하면 COM(Component Object Model)과 같이 Windows 프로그람작성자가 알아두어야 할 항목이 대단히 많다. Web 의 개발도 Windows 의 환경을 확장하는 요인으로 되고 있다. 그러나 걱정할 필요는 없다. 장래에 Windows 2000 프로그람작성기술이 어떻게 확장된다고 해도 이 책을 읽어주신 여러분들에게는 새로운 기술에 추종할 만한 확고한 기초지식이 준비되여 있기때문이다.

색 인

D

E

F

G

H

I

K

L

M

S

ㅈ

ㅊ